PISCINE DE CPP/C++

Table des Matières :

1. Espace de Noms / *NameSpace* :

Définition	Syntaxe

Namespace / Espace de noms :
Un espace de noms, ou namespace, est un **ensemble des symboles, des variables, des fonctions, etc. qui ont un rapport sémantique entre eux.** Par exemple, Le namespace standard, **std::**, est la bibliothèque standard de CPP/C++.
	namespace name {
	// variables;
	}

Aliasing de namespace : Un aliasing de spacename est la **définition d'un nouvel espace de noms égale à un autre espace de nom**.	namespace new_namespace = old_namespace
Remarques	
La notion de namespace introduit un nouvel opérateur appelé l'**opérateur de résolution de portée**, représenté par **"namespace name::"** qui va nous permettre d'accéder à des variables au sein d'un namespace. La syntaxe d'une variable ou fonction globale peut parfois être un petit peu gênante dans le cas où on utilise beaucoup de namespace, pour cela on peut rajouter l'opérateur de résolution de portée sans une "namespace name" qui le précède.	

2. Stdio Streams, Gestion d'Entrées et de Sorties :

En CPP/C++, nous avons à notre disposition des objets qui sont **Cin et Cout** qui respectivement correspondent à **l'entrée et la sortie standard (#include <iostream>)**. Ainsi, nous pouvons récupérer ou écrire plus aisément dans ces dernières. De plus, CPP/C++ met à notre disposition **deux nouveaux opérateurs qui sont double chevrons gauche et doubles chevrons droit (">>" et "<<") qui vont nous permettre de gérer ces flux de données.**

Example	
std::cout << "Hello World !" << std::endl	Cette ligne se lit : dans la sortie standard, je souhaite y injecter la chaîne de caractères "Hello World !" suivi d'un retour à la ligne.
std::cin >> buff :	Cette ligne va nous permettre de récupérer une donnée depuis l'entrée standard et de la stocker dans un buff prédéfinie.
std::cout << "You entered [" << buff << "]" << std::endl :	Cette ligne se lit : Je redirige dans la sortie standard la chaîne de caractères spécifié suivit de celle récupérée précédemment et d'un retour à la ligne.

3. CPP/C++, Un Langage Orienté Objets :

<table>
<tr><th colspan="2">Classe</th></tr>
<tr><td>Syntaxe

class Sample {
public:
Sample(void);
~Sample(void);
};</td><td>Pour chaque classe, nous devons déclarer un constructeur et destructeur de classe, ayant respectivement pour nom le nom de la classe et le nom de la classe précédé d’une tilde.
Plus précisément, lorsqu’une classe est instanciée la fonction de construction (Sample(void)) va être appelée et quand elle est détruite la classe fera appelle à la fonction de destruction (~Sample(void)).</td></tr>
<tr><th colspan="2">Instance</th></tr>
<tr><td>int main(){
Sample instance;
return (0);
}</td><td>Une instance est une instance de la classe Sample. En instanciant ainsi la classe nous faisons automatiquement appelle au constructeur (à la fonction de construction de la classe Sample).
Quand la fonction se termine, ici main(), toute les variables locales vont être détruites, et c’est justement a cette instant que le destructeur (à la fonction de destruction de la classe Sample va être appelé).</td></tr>
<tr><th colspan="2">Constructeur</th></tr>
<tr><td colspan="2">Sample::Sample(void) {
std::cout << “Constructor Called” << std::endl;
return;
}</td></tr>
<tr><th colspan="2">Destructeur</th></tr>
<tr><td colspan="2">Sample::Sample(void) {
std::cout << “Destructor Called” << std::endl;
return;
}</td></tr>
</table>

4. Attributs Membres et Fonctions Membres :

Un **attribut membres** est tout simplement une **variable que nous allons avoir dans notre classe et que nous pourrons utiliser à partir d'une instance. Une fonction membres**, similaire à une attribue membres, est **une fonction qui va appartenir à une classe et que nous pourrons utiliser depuis l'instance de ma classe.**

Contrairement au C, ou pour avoir accès à une fonction à partir d'une structure il fallait utiliser un pointeur sur fonction, en C/C++, nous pouvons directement déclarer une fonction dans notre classe. L'implémentation de cette fonction va fonctionner sur le même modèle que le constructeur et destructeur d'une classe, sauf que cette fois-ci il y aura un type de retour :

Example

void Sample::bar(void) {
  std::cout << "Member Function bar Called" << std::endl;
  return ;
}

5. Pointeur Spécial d'Instance, This :

Le mot-clé "this" va nous **permettre d'ajouter des attributs**, c'est-à-dire des variables et des fonctions qui vont se trouver dans une classe. Plus précisément, "this" va nous permettre de **faire référence à l'instance courante d'une classe au sein du code de la classe en question.** En réalité, "this", en CPP/C++ est un **pointeur** qui pointe sur l'instance courante. Contrairement au C, avec "this" nous pouvons directement faire appelle a une fonction de notre instance.

Example	

classe Sample {
  public :
  int foo;
  sample(void);
  ~sample(void);
  void bar(void);
};
	Sample::Sample(void) {
	  std::cout << "Constructor Called" << std::endl;
	  this->foo = 42;
	  std::cout << "this->foo" << this->foo << std::endl;
	  this->bar;
	  return;
	}

Remarques	
Si notre classe contient des attributs, il est intéressant de venir initialiser ces derniers dans le constructeur. De plus, puisque "this" est un pointeur, on va utiliser la notation **"this->"** et non "this.".	

6. Liste d'Initialisation :

En ce qui concerne les constructeurs qui prennent des paramètres et l'initialisation des différents attributs qui vont correspondre, il existe deux manières de procéder :

<table>
<tr><th colspan="2">Example</th></tr>
<tr><td rowspan="4">Pour la classe suivante :

classe Sample {

public :
char a1;
int a2;
float a3;
sample(char p1, int p2, float p3);
~sample(void);
};</td><td>Initialisation Manuelle</td></tr>
<tr><td>Sample::Sample(char p1, int p2, float p3) {
std::cout << “Constructor Called” << std::endl;
this->a1 = p1;
std::cout << “this->a1” << this->a1 << std::endl;
this->a2 = p2;
std::cout << “this->a2” << this->a2 << std::endl;
this->a3 = p3;
std::cout << “this->a3” << this->a3 << std::endl;
return;
}</td></tr>
<tr><td>Liste d'Initialisation</td></tr>
<tr><td>Sample::Sample(char p1, int p2, float p3) : a1(p1), a2(p2), a3(p3) {
std::cout << “Constructor Called” << std::endl;
std::cout << “this->a1” << this->a1 << std::endl;
std::cout << “this->a2” << this->a2 << std::endl;
std::cout << “this->a3” << this->a3 << std::endl;
return;
}</td></tr>
<tr><th colspan="2">Remarques</th></tr>
<tr><td colspan="2">Il est conseillé d'utiliser la liste d'initialisation et non l'initialisation manuelle.</td></tr>
</table>

7. Constante, const :

L'utilisation d'une constante en CPP/C++ est multiple et bien plus importante qu'en C :

- Dans un premier temps, l'utilisation de "const" permet de **rendre un attribut constant.** En C, pour initialiser un constante il fallait faire la déclaration et l'assignation sur la même ligne (int const = 42). En CPP/C++ une constante peut être **déclarée et initialisée sur différentes lignes grâce à l'utilisation d'une liste d'initialisation.**

- Dans un second temps, le mot-clé "const" entre la parenthèse fermante et soit le point virgule pour la déclaration soit l'accolade ouvrante pour la définition d'un prototype, **permet de signifier au compilateur que cette fonction membre de modifiera jamais l'instance courante.**

8. Visibilité, "public" et "private" :

"Public" et "Private" vont nous permettre de **contrôler l'encapsulation des membres de notre classe**, à savoir, les attributs et les fonctions membres qui seront visibles uniquement depuis l'intérieur de la classe bien de l'extérieur. Cela va nous permettre de choisir ce que nous voulons garder public et ce que nous voulons garder privé au sein de notre classe.

	Public	Privé
Définition	**Accessibles depuis l'intérieur et l'extérieur de la classe**	**Accessibles que depuis l'intérieur de la classe.**
Syntaxe	"public:"	"private:"
Remarque		
D'après les conventions les plus répandues, les noms des attributs et fonctions membres privées doivent contenir un "_".		

Attention, **le constructeur et le destructeur de ma classe doivent être publics** tout simples parce qu'elles sont primordiales pour pouvoir construire et détruire ma classe.

9. Classe VS. Structure:

En CPP/C++, il existe également ce que nous appelons des structures, sa syntaxe est la même que celle des classes sauf qu'au lieu d'utiliser le mot-clé "class" nous utilisons "struct".Leur différence, réside dans le fait qu'**une structure à un scope par défaut public** et qu'**une classe un scope par défaut privé**. Ainsi, la structure n'a pas grand intérêt en structure.

10. Les Accesseurs:

Les accesseurs sont utilisés pour savoir et/ou influer sur les attributs et fonctions membres privés d'une classe. Plus précisément, ce sont des **fonctions proxis** qui vont faire l'intermédiaire entre l'utilisateur et les membres privés. Les conventions indiquent que ces fonctions proxis devrait, en règle générale, prendre les préfixes **"get" et "set"**, qui vont respectivement p**ermettre de récupérer la valeur d'un attribués membres** (const) **et de "seter" la valeur d'un attribut.**

Example	

class Sample {
public:
sample(void);
~sample(void);
int getfoo(void)const:
void setfoo(int v):
private:
in foo;
};
	int Sample::getfoo(void)const {
	return (this->foo);
	}
	
	void Sample::setfoo(int v) {
	if (v >= 0)
	this->foo = v;
	return ;
	}

11. Comparaison :

En CPP/C++, on va pouvoir **utiliser les fonctions membres pour ajouter à nos classes une fonction "compare" qui va nous permettre de faire une comparaison structurelle sur deux instances afin de vérifier une égalité ou non.**

Cette fonction "compare" va prendre en paramètre l'adresse d'une instance de ma classe et faire une comparaison entre l'instance courante et celle passée en paramètres.

Example	

class Sample {
public:
sample(void);
~sample(void);
int getfoo(void)const:
void compare(int v):
private:
in foo;
};
	int Sample::compare(sample *other)) const {
	if (this->foo < other->getfoo())
	return (-1);
	else if (this->foo > other->getfoo())
	return (1);
	return (0);
	}

Attention, deux instances sont physiquement égales si leurs deux adresses sont les mêmes, et deux instances sont structurellement égales si leurs attributs sont les mêmes.

12. Attributs et fonctions non membres :

Nous avons déjà vu qu'un attribut ou/et une fonction membres désignent quelque chose qui existe au niveau de l'instance de notre classe (c'est-à-dire qu'il faut instancier la classe pour que l'on puisse utiliser l'attribut et/ou la fonction en question). De fait, lorsqu'on modifie les attributs et/ou fonctions membres d'une instance, ceux d'une autre resteront inchangés. **Des attributs et/ou fonctions non membres vont pouvoir être modifiés au niveau de la classe et non au niveau de l'instance.** Leur syntaxe, reprend un mot-clé courant en C : **"static."**

13. Pointeurs sur Membres :

Les pointeurs en CPP/C++ sont accompagnés de 2 nouveaux amis : les pointeurs sur membres et les pointeurs sur fonctions membres :

<table>
<tr><td colspan="2">Pointeurs sur Membres</td></tr>
<tr><td>int Sample::*p = NULL;</td><td>p = &Sample::foo;</td></tr>
<tr><td>En ajoutant "Sample", le nom de la classe, "::" avant l'étoile "*", je déclare non plus un simple pointeur sur int mais un pointeur sur un entier qui est membre de la classe "Sample."</td><td>Ensuite, je vais pouvoir assigner l'adresse de mon membre "foo" au pointeur en question. C'est-à-dire que "p" est un pointeur qui prend l'adresse d'un entier qui est membre de la classe "Sample."</td></tr>
<tr><td colspan="2">instance.*p = 21; // instancep->*p = 21</td></tr>
<tr><td colspan="2">A présent, on va pouvoir assigner une nouvelle valeur à notre attribut "foo" en utilisant le pointeur. Pour cela, nous allons introduire un nouvel opérateur ".*". Plus précisément, cela signifie que nous allons assigner la valeur 21 au contenu du pointeur "p" de l'instance choisie. Attention, pour les instances de piles je vais utiliser l'opérateur ".*" et l'opérateur "->*" pour les pointeurs.</td></tr>
<tr><td colspan="2">Pointeurs sur Fonctions Membres</td></tr>
<tr><td>void (Sample::*f)(void) const;</td><td>f = &Sample::bar;</td></tr>
<tr><td>La syntaxe d'un pointeur sur fonctions membres en CPP/C++ se rapproche de celle utilisée pour pointer sur une fonction en C. La différence réside dans le fait que je vais spécifier à quelle classe je fais référence, soit "Sample::".</td><td>Ici, je dis que mon pointeur "f" va pointer vers la fonction membre "bar" de ma classe "Sample."</td></tr>
<tr><td colspan="2">(instance.*f)(); / (instance->p*f)();</td></tr>
<tr><td colspan="2">De nouveau, on ne sais pas encore à quelle instance je fais référence, de fait nous allons de nouveau utiliser l'opérateur ".*" et "->*" pour spécifier l'instance avec laquelle j'utiliser mon pointeur sur fonction membres.</td></tr>
</table>

14. New and delete :

L'opérateur new et son homologue delete, utilisés dans l'allocation mémoire, vont remplacer malloc et free en C. En CP/P, l'utilisation de malloc est possible mais très déconseillé : l'appel du constructeur est du destructeur n'est pas faîtes lorsqu'on alloue des objets ou des classes.

New va allouer la classe ou l'objet et appelé son constructeur, tandis que delete va appeler son destructeur et free. Attention, on ne peut utiliser free sur un objet ou un classe qu'on a new, de même on delete pas un objet ou une classe qu'on a mallox.

Example

```
classe Student {
    private:
        std::string _login;
    public:
        Student(str::string login) : _login(login);
            std::cout << "Student " << this->_login << " is born" << std::endl;
        ~Student()
            std::cout << "Student " << this->_login << " died" << std::endl;
}
```

```
int main() {
    Student    bob = Student("mboy");
    Student*   jim = new Student("mla-rosa");

    // do somthing;
    delete jim;
    return (0);
}
```

Pour allouer des tableaux de valeurs

```
int main() {
    Student     *students = new
                            student[42];

    // do somthing;
    delete [] students;
    return (0);
}
```

15. Références :

En C, on utiliser les pointeurs pour pouvoir manipuler une variable en dehors de notre scope ou lorsque l'on avait besoin d'allouer une variable sur le tas et nos sur le stack (en utiliser les opérateurs "*" et "&". **Les références sont un alias vers une variable existante, ou plus précisément, sont des pointeurs qui sont constant, toujours *déréférencé, et jamais nul.** Plus simplement, une référence est un pointeur constant qui pointe toujours sur la même chose. Attention, les références complètent les pointeurs, elles ne les remplacent pas. Même si syntaxiquement plus pratiques, elles ne peuvent pas être nulles..

<table>
<tr><th colspan="2">Example</th></tr>
<tr><th>Programme</th><th>Remarques</th></tr>
<tr><td>

```
int    main() {
   int     numberOfBalls = 42;
   int*   ballsPtr = &numberOfBalls;
   int& ballsRef = numberOfBalls;

   std::cout << numberOfBalls << " " << *ballsPtr
                        << " "   << ballsRef << std::endl;
   *ballsPtr = 21;
   std::cout << numberOfBalls << std::endl;
   ballsRef = 84:
   std::cout << numberOfBalls << std::endl;
   return (0);
}
```

</td><td>Je ne peux pas : int &ballsRef;
Pourquoi ? Car cela revient à créer une référence qui ne pointe sur rien alors qu'une référence ne peut être égale à nul.

De plus, je n'ai pas besoin d'inclure l'opérateur "&" lorsque je veux utiliser ma référence puisqu'une référence est toujours déréférencé</td></tr>
<tr><th colspan="2">Output</th></tr>
<tr><td colspan="2">42 42 42 // numberOfBalls // son pointeur *ballsPtr // sa référence ballsRef
21 // Modification de numberOfBalls en passant par son pointeur *ballsPtr
84 // Modification de numberOfBalls en passant par sa référence ballsRef</td></tr>
<tr><th colspan="2">Passage de Paramètre par Référence & Retour par Référence</th></tr>
<tr><td colspan="2">En C, on pouvait utiliser des pointeurs en tant que paramètre de fonctions. En CPP/C++, on peut également utiliser des références en tant que paramètre ou retour (une variable qui prend la valeur le retour par référence d'une fonction est une référence donc elle est constante, toujours *déréférencée, et jamais nulle).</td></tr>
</table>

Déréférencer un pointeur c'est mettre l'opérateur "" pour dire qu'on est intéressé par ce qui est pointé et non pas le pointeur.

16. Filestream :

On a vu que l'on pouvait utiliser les streams pour agir avec la sortie, l'entrée standard, la sortie d'erreur. Maintenant, on peut également **utiliser les streams pour agir sur les fichiers.** Par exemple, je peux utiliser std::ifstream ou std::ofstream pour créer un input ou output file stream. Pour lire le contenu des ses fichiers je vais utiliser l'opérateur "<<".

Example

```
int main() {
   std::ifstream    ifs("numbers"); //je créer un input file appelé numbers
   unsigned int    dst;
   unsigned int    dst2;

   ifs >> dst >> dst2;
   std::cout << dst << " " << dst2 << std::endl;
   ifs.close();

   std::ofstream    ofs("test.out");
   ofs << "Je m'appelle Melody BOY" << std::endl;
   ofs.close();
}
```

17. Polymorphisme Ad-hoc (surcharge de fonction) :

La surcharge de fonction est tout simplement un principe de CPP/C++ qui va nous permettre de **définir plusieurs fonctions ayant le même nom mais qui prennent des paramètres différents** - chose impossible en C. La surcharge de fonction n'est pas réservée aux fonctions membres d'une classe, elle peut être appliquée pour tout type de fonction.

Plus précisément, la surcharge de fonction va nous permettre de **spécialiser le traitement d'une fonction en fonction de ses entrées.**

Example

```
class Sample {
   public:
      Sample(void);
      ~Sample(void);

      void      bar(char const c) const; // 1ère surcharge
      void      bar(int const n) const; // 2ème surcharge
      void      bar(float const z) const; // 3ème surcharge
      void      bar(Sample const& l) const; // 4ème surcharge
}
```

18. Surcharge d'Opérateur:

Le CPP/C++ est u**n langage qui propose une syntaxe afin de pouvoir étendre les opérateurs de CPP/C++, afin de pouvoir ajouter de la sémantique et de la syntaxe au langage pour l'adapter à nos besoins.**

Pour commencer, quelques vocabulaire :

- **Notation Infixe** : Opération arithmétique de type 1 + 1, ou l'opérateur "+" se situe entre deux deux nombres (1 et 1);
- **Notation Préfixe ou Notation fonctionnelle** : Opérateur arithmétique de type = 1 1 qui peut être traduit en =(1, 1), qui s'apparente à une fonction et qui peut être remplacé par une fonction.
- **Notation Postfixe** : Opérateur Arithmétique de type 1 1 =, utilisé dans le calcul de pile.

Example	Remarque

class Integer {
  public:
  Integer(int const n);
  ~Integer(void);
  int getValue(void) const;
  integer& operator=(Integer const& rhs);
  integer operator+(Integer const& rhs) const;
 	
  private:
  int _n;
}
 	
Integer& Integer::operator=(Integer const& rhs) {
  this->_n = rhs.getValue();
  return (*this);
}
 	
Integer Integer::operator+(Integer const& rhs) const {
  return (Integer(this->_n + rhs.getValue());
}
 	
std::ostream& operaor<<(std::ostream& o, Integer const &rhs) {
  o << rhs.getValue();
  return (o);
}
	Le mot clé "operator" va permettre de faire d'une simple déclaration de fonction membre de ma classe une surcharge d'opérateur, en spécifiant l'opérateur voulu entre le mot clé et la parenthèse ouvrante.
	
	L'opérateur d'assignation, par sa définition, est binaire où on a à gauche ce dans quoi je veux assigner et à droite ce que je veux assigner.
	
	Le dernier exemple, avec l'opérateur "<<", va nous donner la possibilité de pouvoir modifier le code de l 'OStream (dans l'exemple ci-dessous).

Attention	
La surcharge d'opérateur peut vite devenir bordélique, pour cela il faut suivre certaines **règles strictes**: - Il faut que la surcharge d'un opérateur soit **naturelle** ; - Il faut que la surcharge d'un opérateur est un **rapport avec la sémantique naturelle de ce derniers** ; - Il faut **éviter** d'en faire, tout simplement.	

19. Forme Canonique :

La forme canonique est **une façon de construire des classes de bases afin de s'assurer une interface raisonnable et surtout identique sur tout notre travail.** A partir de là, n'importe quelle personne qui va lire nos sources ou quand nous irons nous-même lire nos sources, on aura la conviction qu'il y aura toujours un certain nombre de f**onctions disponibles dans toute notre classe permettant un traitement uniforme.**

Une classe est dit canonique quand elle contient au moins :

- **Un constructeur par défaut**, le constructeur par défaut peut être dans la section privée de la classe ainsi de l'extérieur on ne pourra pas faire d'instanciation par défaut mais il sera écrit clairement dans la déclaration qu'un constructeur par défaut est bien présent mais qu'il n'est pas disponible pour les utilisateur ;
- **Une constructeur par copie** (c'est-à-dire un constructeur prenant en paramètre une instance de la classe que l'on est entrain de déclarer afin de pouvoir réaliser une copie de cette classe) **;**
- **Un opérateur égale/assignation ("=") qui permettra donc de faire une assignation à partir d'une autre instance de cette classe** (qui se rapproche du constructeur par copie, sachant que dans le cas du constructeur par copie on a bien une nouvelle instance de créer tandis que dans l'opérateur d'assignation on a tout simplement une mise à jour de l'instance courante;
- **Un destructeur.**

Example

```
class Sample {
   public:
      Sample(void);   // Constructeur 1
      Sample(int const n); // Constructeur 2
      Sample(Sample const& src);  // Constructeur 3
      ~Sample(void); // destructeur
      Sample&       operator=(Sample const& rhs); // Surcharge d'opérateur
      int               getFoo(void) const; // Accesseur

   private:
      int       _foo; // Entier
}
```

20. Héritage :

L'héritage, en quelques mots, est simplement **un moyen de factoriser des comportements d'une série de classes différentes qui ont des comportements très similaires, et de les rassembler dans une même classe pour ensuite permettre aux objets dérivés d'utiliser ces comportements là.**

"Protected" est un nouveau niveau d'encapsulation, utilisé uniquement dans le cadre de l'héritage, avec qui va se rajouter à "public" et "private". "Public" signifie que c'est accessible de n'importe où, **"protected" signifie ça veut dire que c'est accessible uniquement par une instance de la classe ou une instance d'une classe dérivée** et "private" signifit que ça peut être accessible que depuis une instance de la classe.

Plus simplement, dans le cadre de l'héritage, une élément "public" peut accéder de n'importe où, un élément "private" est accessible uniquement depuis la classe de l'héritage et non pas depuis les classes qui en héritent et un élément "protected" est accessible depuis la classe de l'héritage et depuis les classes qui en héritent..

Attention, une classe peut hériter de plusieurs classes, dans ce cas on appelle cela un héritage multiple.

<table>
<tr><th colspan="2">Example</th></tr>
<tr><td>

```
Class Cat: public Animal {
   private:
      int    _numberOfLegs;

   public:
      Cat(void);
      Cat(Cat const& );
      Cat&    operator=(Cat const&);
      ~Cat(void);
      void    run(int distance);
      void    scornSomeone(std::string const&
                                    target);

}
```

</td><td>

```
Class Dog: public Animal {
   private:
      int    _numberOfLegs;

   public:
      Dog(void);
      Dog(Dog const& );
      Dog&    operator=(Dog const&);
      ~Dog(void);
      void    run(int distance);
      void    barkSomeone(std::string
                          const& target);

}
```

</td></tr>
<tr><td>

```
class Animal {
   private:
      int    _numberOfLegs;

   public:
      Animal(void);
      Animal(Animal const& );
      Animal&    operator=(Animal const&);
      ~Animal(void);
      void    run(int distance);
      void    call(void):
}
```

</td><td>

Remarque

Ici, les classes Cat et Dog vont hériter des fonctions membres de la classe Animal comme la fonction run et call dans l'exemple ci-dessous.

On peut également faire un héritage en utilisant "private" et "protected" et non pas seulement avec "public" : ce qu'on appelle **les niveaux d'encapsulation de l'héritage**

</td></tr>
</table>

21. Polymorphisme par sous-typage (virtual) :

Petit rappel, on a vu que **hériter d'une classe veut dire qu'on est dans cette classe en question.** De plus, on peut **redéfinir dans une classe fille un comportement d'une classe mère.** Dans ce cas, on dit qu'on va "**overrider**" une fonction.

Grâce à l'utilisation du mot clé **"virtual"**, **la résolution de l'appelle de fonctions va devenir dynamique et non statique** (elle sera faite à l'exécution et non à la création). Plus précisément, une fonction membre virtuelle est appelée une "**méthode**."

Example

```
class Character {
   public:
      void    sayHello(std::string const& target);
};

class Warrior: public Character { //Warrior est un Character Warrior)
   public:
      virtual void    sayHello(str::string const& target);
};

void    Character::sayHello(std::string const& target) {
   std::cout << "Hello " << target << " !" << std::endl;
}

void    Warrior:sayHello(std::string const& target) {
   std::cout << "F**** off " << target << ", I don't like you !" << std::endl;
}

int    main(void) {
   Warrior*      a = new Warrior(); // OK, Normale
   Character*   b = new Warrior(); //OK,  Warrior is a Character
   Warrior*      c = new Character //KO,  Character is not a Warrior

   a->sayHello("students"); //Utiliser de la fonction membre de la class Warrior
   b->sayHello("students"); //Utiliser de la fonction membre de la class Warrior et non de
                                                                  Character grâce au mot-clé "virtual)
}
```

22. Classes Abstraites et Interfaces :

Une méthode dite “pure”, **initialisée avec “= 0” lors de la définition**, ne peut être implémenter (puisqu’elle est égale à zéro) et ne peut instancier (la classe aura alors un comportement qui ne sera pas là, elle va devenir abstraite). Par convention, toutes les classes abstraites prennent le **préfixe “A.”** En d’autres termes, une classe abstraite à des comportements définis et d’autres qui ne le sont pas. Sans oublier, qu’une classe abstraite ne peut pas être instanciée. **Pour pouvoir instancier une classe qui hérite d’une classe abstraite, va devoir implémenter les comportements définis comme “abstraits” dans la classe parente.** Sinon, ces comportements restent abstraits.

De même, au lieu de mettre simplement un comportement abstrait et un comportement concret, je vais pouvoir **mettre que des comportements abstraits** : alors, on créera une **“interface”**. Par convention, une interface prendra le préfixe “I”. Attention, une interface a également **aucun attribut.** Plus spécifiquement une interface sert pour utiliser des objets dérivés : c’est simplement une manière de définir un comportement.

Example

```
class ACharacter {
   public:
      virtual void     attack(std::string const& target) = 0;
      void             sayHello(std::string const& target):
};

class Warrior {
   public:
      virtual void     attack(std::string const& target); //Implémentation comportements
};                                                                                    abstraits

class ICoffeeMaker {
   public:
      virtual void       fillWaterTank(IWaterSource* src) = 0;
      virtual ICoffee  makeCoffee(std::string const& type) = 0;
};

void    ACharacter::sayHello(std::string const& target) {
   std::cout << “Hello ” << target << “ !” << std::endl;
}

void    Warrior:sayHello(std::string const& target) {
   std::cout << “F**** off “ << target << “, I don’t like you !” << std::endl;
}

int    main(void) {
   ACharacter*       a = new Warrior();
   ACharacter*       b = new ACharacter(); // KO, ACharacter est  abstrait

   a->sayHello(“student”);
   a->attack(“roger”);
}
```

Remarque
Quand une classe est abstraite, ça veut dire que mes classes qui héritent de cette classe devront avoir leurs comportements respectifs (par exemple, ici on définit qu'un Character devrait pouvoir attaquer, il ne pourra pas attaquer par lui-même. Un Character en lui-même n'a pas d'attaque, c'est juste une manière de factoriser certains comportement comme la manière de dire bonjour (sayHello). Cependant, les classes filles, elles, devront avoir le comportement "attack" même si la classe mère en elle-même n'en à pas.

23. Classes Imbriquées :

Une classe imbriquée veut dire qu'on peut **définir une classe dans la définition d'une autre classe.** Cela fonctionne un peu comme les namespaces, du fait de son opérateur "::".

Exemple			
			Remarques

class Cat {
public:
class Leg {
//[...]
}
};
	class Dog {
	public:
	class Leg {
	//[...]
	}
	};
		int main(void) {
		Cat somecat;
		Cat::Leg somecatsleg;
		return (0);
		}
			L'imbrication de classe permet d'avoir deux classes "Leg" séparés dans deux classes différentes (Cat et Dog ici).

24. Les Exceptions :

Les exceptions sont **une nouvelle façon de gérer les erreurs.** Plus précisément, les exceptions sont une manière de remonter un message à travers la pile d'appels quand on trouve une erreur. Au lieu de gérer les erreurs en renvoyant des “-1”, des “-2”, etc. comme on le faisait en C, on pourra utiliser les exceptions.

Le concept d’exception s’accompagne des expressions **try, throw et catch** :

- **Try :** va nous permettre d’**essayer de faire quelque chose qui pourra renvoyer une exception ;**
- **Throw : va lancer une exception** en utilisant le **type “std::exception”** qui devra être la base de toutes exceptions renvoyées. Plus précisément, **throw va remonter dans les blocs jusqu’à trouver un bloc catch.**
- **Catch :** va venir **“catcher” une “std::exception”,** ici appelée “e”, **et la traiter**.

Attention, comme leurs noms l'indiquent, **une exception devrait rester exceptionnelle**. Par celà, on veut dire qu’une fonction qui foire une fois sur deux ne devrait pas renvoyer une exception. En effet, si on appelle souvent cette fonction, cela va utiliser des ressources inutiles dans tous les sens. Le fait de “throw” une exception c’est quelque chose qui est relativement coûteux en ressources, en tout cas beaucoup plus coûteux en ressources que de simplement renvoyer une valeur d’erreur. Par contre, dans le cas ou ca ne renvoie pas souvent une erreur, et dans le cas ou c’est quelque chose qui ne fait pas partie du fonctionnement normal de programme, qui est donc exceptionnelle, l’utilisation d’une exception est grandement conseillée (pleins de fonction system comme new renvoient des exceptions).

<table>
<tr><th colspan="3">Exemple Basic</th></tr>
<tr><td>

```
void test1(void) {
 try {
    //Do some stuff here
    if (/* there's a error */)
      throw std::exception();
    else
       //Do some more stuff;
 }
 catch (std::exception e)
    //Handle the error here
}
```

</td><td>

```
void test2(void) {
      //Do some stuff here
      if (/* there's a error */)
        throw std::exception();
      else
         //Do some more stuff;
}
```

</td><td>

```
void test3(void) {
 try
    test2();
catch (std::exception& e)
  //Handle the error here
}
```

</td></tr>
<tr><th colspan="3">Remarques (1)</th></tr>
<tr><td colspan="3">Une fonction de gestion d’exception n’a pas obligatoirement besoin d’expression try et catch, mais la fonction qui appelle cette dernière doit posséder un try et un catch, comme démontrer pour test2 et test3 ci-dessus. Je peux également catcher mon exception par référence.</td></tr>
<tr><th colspan="3">Exemple Complex</th></tr>
</table>

```
void test4(void) {
  class PEBKACExecptoion : public std::exception {
    public:
      virtual const char* what(void) const throw()
        return (“Problem exists between keyboard and chair”);
  };
  try
    test(3);
  catch (PEBKACExecptoion& e) //Catch Spécifique
    // Handle the fact that the user is a idiot
  catch (std::exception& e) //Catch Générique
    //Handle other exceptions that are like std::exception
}
```

Remarques (2)

Ici, je vais définir une nouvelle exception intitulée “PEBKACException” héritant publiquement de “std::exception” et qui indique qu’un problème se trouve entre la chaise et le clavier. Ainsi, “PEBKACException” est une “std::exception” que je peux donc à présent manipuler.

Je définis la méthode “what” qui est une virtual const char* et qui “throw” rien du tout. Dans cet exemple, le “throw” est un **“throw spécifier”** : ce qui signifie que cette fonction peut “throw" quelque chose ou pas à la fin de la définition de cette fonction. Plus simplement, c’est une **manière très propre de préciser quelles sont les exceptions qu’une méthode peut remonter.**

En plus de “catch” une “PEBKACException” si l’exception est de ce type la fonction va renvoyer toute autre exception qui hérite de “std::exception” ou qui est “std::exception” : appelé un catch générique (tout autre “catch” sera appelé un “catch spécifique”).

25. Cast - Introduction et Vocabulaire :

Contrairement en C, le CPP/C++ propose deux types de “cast” :

- Le **cast explicite** où l’on met entre parenthèse le type vers lequel on veut “caster”;
- Le **cast implicite** où l’on rien besoin d’ajouter au code pour le “casting” se fasse.

Ces deux types de “cast” **produisent un certain type de comportement qui n’est pas toujours le même,** d’où la difficulté des “casts” en CPP/C++.

Petit rappel, **les “casts” vont nous permettre de convertir des types vers d’autres types et pour cela on allait avoir un réarrangement des bits pour pouvoir correspondre au nouveau type.**

Il existe une action de “cast” qui est la **réinterprétation** qui nous permet de **réinterpréter des adresses d’un type plus précis à un type plus général et inversement, ce qui peut parfois poser des problèmes.** Une réinterprétation d’un type plus précis à un type plus général est une **démotion : une réinterprétation descendante** (qui peut “caster” explicitement ou implicitement)**.** À l'inverse, une réinterprétation d’un type plus général à un type plus précis est une **promotion : une réinterprétation ascendante** (qui nécessite un “cast” explicite).

a. Cast de type conversion :

Le codage des entiers et des doubles est très différent, et pourtant, dans l’exemple ci-dessous, les valeurs seront égales. Cette notion de **transformer le codage d’un octet vers un autre codage pour garder une même valeur** s’appelle une conversion. Ainsi, il est possible de faire des conversions dites implicites mais aussi de faire une conversions dites explicite.

Exemple

```
int    main(void) {
   int         a = 42;    // Valeur de référence
   double    b = a;     // Implicit conversion const
   double    c = (double)a;     // Explicit conversion const
   double    d = a; //Implicit conversion -> Ok
   int         e = d; //Implicit démotion -> Ko
   int         f = (int)d; // //Explicit démotion -. Ok
   dump._32bits_interger(a); //Représentation binaire entier
   dump._64bits_double(b); //Représentation binaire double
   dump._64bits_double(c);
   dump._64bits_double(d);
   dump._32bits_interger(e);
   dump._32bits_interger(f);
   return (0);
}
```

Output
[42]: 00000000 00000000 00000000 00101010 [42.000000]: 0 10000000100 0101 0000000000000000000000000000000000000000000000000000 [[42.000000]: 0 10000000100 0101 0000000000000000000000000000000000000000000000000000 [42.000000]: 0 10000000100 0101 0000000000000000000000000000000000000000000000000000 [42]: 00000000 00000000 00000000 00101010 [42]: 00000000 00000000 00000000 00101010
Remarques
La notion de **hiérarchie des types** implique que je peux faire une conversion de type implicite d'un entier vers un double mais pas d'un double vers un entier (puisque les doubles sont plus précis et donc hiérarchiquement supérieure aux entiers). Cependant, je peux faire une conversion de type explicite qu'on appellera une démotion. Concrètement, cela est possible mais il y aura alors une perte de précision puisque qu'on passe d'un type double de 64 bits à à un type entier de 32 bits.

b. Cast de type reinterpretation:

Un cast de type réinterprétation est un **“cast” dit identitaire.** Dans ce cas, **la valeur de base après conversion garde les mêmes bits dans le même ordre** (de fait, il n’y pas réellement eu conversion pendant le “cast”).

Exemple

int main(void) {
  float a = 420.042F; // Valeur de référence
  void* b = &a; // Implicit reinterpretation cast
  void* c = (void *)&a; // Explicit reinterpretation cast
  void* d = &a; // Implicit promotion -> OK
  int* e = d; // Implicit démotion -> KO (hasardeux)
  int* f = (int *)d; // Implicit démotion -> OK
 	
  printf(“%p, %f\n”, &a, &);
  printf(“%p\n”, b); /
  printf(“%p\n”, c);
  printf(“%p\n”, d);
  printf(“%p, %d\n”, e, *e);
  printf(“%p, %d\n”, f, *f);
  return (0);
}

Remarques
Dans le cas des types d’adresse nous allons retrouver le même genre de **hiérarchie** que celle vu précédemment. Ici, le type “void *” est plus intéressant que les autres types d’adresse. Effectivement, ce type va nous permettre de contenir une adresse sur n’importe quel type. A partir de cela, nous pouvons dire que **le type “void *” est plus général que tous les autres types d’adresse.** Par exemple, le type “float *” sera un type d’adresse plus précis que le type “void *.” Ainsi nous retrouvons la notion de hiérarchie des types et les mêmes implications que dans le cas d’une conversion pour une réinterprétation.

c. Cast de type Upcast et Downcast :

Attention, ici nous allons utiliser les “casts” en C afin d’expliquer comment utiliser les “casts” en CPP/C++. Bien évidemment, par la suite, nous allons utiliser que les “casts” en CPP/C++.

Précédemment, nous avons vu qu’il existait **une action de “cast” qui est la réinterprétation** qui nous permet de réinterpréter des adresses d’un type plus précis à un type plus général et inversement, ce qui pouvait parfois poser des problèmes. Les différences en précisions mènent au **concept de hiérarchie de type**.

Ici, dans le contexte du CPP/C++ nous allons introduire une **nouvelle hiérarchie de type généré par l’existence des classes et des diverses interactions possibles entre les adresses de ces dernières. Cette hiérarchie**

On sait que **les pointeurs sont compatibles entre l’adresse d’une classe de base et celle d’une classe fille**. Ainsi, **nous allons utiliser cette comptabilité des pointeurs pour mettre en lumières les comportements des adresses de classe dans le cas d’une réinterprétation. Et ces interprétations d’adresse de classe en CPP/C++ vont porter le nom de “upcast” et de “downcast ” :**

- **Un “downcast” est une réinterprétation descente, une démotion.**
  Un “downcast” ne peut pas être fait implicitement, le compilateur ne le permettra pas. Ainsi, il faudra **toujours utiliser un cast explicite**.
- **Un “upcast” est une réinterprétation ascendante, une promotion**.
  Un “upcast” peut se faire tant **explicitement que implicitement.**

Exemple

```
class Parent                    {};
class Child1 : public Parent {};
class Child2 : public Parent {};

int    main(void) {
   Child2*    a;  // Valeur de référence
   Parent*    b = &a;  //  Réinterprétation par cast implicite
   Parent*    c = (Parent*)&a;   //  Réinterprétation par cast explicit
   Parent*    d = &a;  //Upcast implicite -> OK
   Child1*    e = d; // Downcast implicite -> KO
   Child2*    f = (Child2*)d; //Downcast explicit -> OK, mais à éviter
}
```

Remarques
Dans l'exemple ci-dessus **il y a naturellement une hiérarchie qui va s'instaurer au sein de nos classes**, c'est-à-dire que la classe "Parent" va être une version plus générique, moins spécialisée, des classes "Child1" et "Child2". A l'inverse, les classes "Child1" et "Child2" seront des versions plus précises, plus spécialisées, de la classe "Parent." Attention, le compilateur ne nous empêche pas de faire un cast explicite entre deux classes de même hiérarchie, comme entre "Child1" et "Child2" dans l'exemple précédent. Par contre, à l'exécution, il y aura des problèmes puisque rien ne me dit que l'implémentation de classe "Child1" est identique à celle de classe "Child2." A part tous les contenues de base de la classe "Parent", les deux classes "Child1" et "Child2" peuvent être aussi différentes que l'on peut imaginer. Ainsi, **le cast entre deux classes hiérarchiquement égales est à éviter.**

26. Cast - Cinq Types :

Depuis la notion de “cast” en CPP/C++, nous avons découvert que **les actions “cast” étaient toutes des conversions mais que certaines conversions avaient des propriétés intéressantes.** Nous avons vu le cas en particulier des **conversions identitaires qu’on appelait des réinterprétations**. Nous avons également un cas particulier des **réinterprétations qui concerne les qualifieurs de types**. Ensuite, nous avons vu qu’en CPP/C++ allait introduire d’autres types de comportements à savoir les **“upcast” et les “downcast”** qui sont également des cas particuliers de réinterprétation mais **dans un contexte d’arbres d’héritage de classe.**

Ainsi, au total nous avons **5 actions de “cast”** possibles :

- **Les changements de types qualifieurs**;
- **Les conversions** ;
- **Les réinterprétations** ;
- **Les “upcasts”** ;
- **Les “downcasts.”**

Ces 5 types de “casts” vont correspondre en CPP/C++ à l’un ou l’autre des différents opérateurs de “cast” mise à disposition pour le langage. Ils seront un certain nombre, que nous allons à présent voir.

a. Static_cast :

Le “static cast” va nous permettre de faire des conversions simples entre des valeurs directes (c’est le “cast” le plus usuel dans ce contexte là) mais qui va également **nous permettre de faire des “upcasts” et des “downcasts” sur une hiérarchie de classe que nous connaissons et empêcher de faire des “casts” entre des hiérarchies de classe qui ne sont pas liées.** Ce “static_cast” est dit statique puisqu’il est **exécuté lors de la compilation** (il est donc statique). Le “static cast” introduit :

- Un **nouveau mot-clé : static_cast** ;
- **Suivi d’un entre-chevron : <…>;**
- **Suivi d’un paramètre entre parenthèses : (...)**;

Puisque nous sommes en CPP/C++ nous allons plus utiliser les “casts” comme nous les aurions utilisés en C, mais utiliser cette nouvelle manière de faire.

Exemple Changements de Types Qualifieurs

```
int    main(void) {
   int          a = 42;  // Valeur de Référence
   double    b = a;  // Promotion Implicite -> OK
   int          c = b; // Démotion Implicite -> KO
   int          d = static_cast<int>(b); // Démotion Explicit -> KO
                        // Je souhaite un “static_cast” de l’expression b vers le type int
   return (0);
}
```

Exemple Upcast et Downcast

```
class Parent                {};
class Child1 : public Parent {};
class Child2 : public Parent {};
class Unrelated             {};

int    main(void) {
   Child1       a;  // Valeur de référence

   Parent*      b = &a;  //  Implicit Upcast ->OK
   Child1*      c = b;   //  Implicit Downcast -> KO
   Child2*      d = static_cast<Child2 *>(b); //Explicit Downcast -> OK, risqué !
   Unrelated* e = static_cast<Unrelated *>(&a); // Explicit Conversion -> KO
 }
```

Remarque

Le premier **"static_cast"** dans l'exemple ci-dessus, qui est **explicite**, vas fonctionner. Mais cela peut poser **problème**, qui va être adressé plus tard (un moyen de l'éviter et l'éliminer).

Le "static_cast" utiliser pour des "upcasts" et "downcast", va pouvoir détecter lorsqu'on souhaite faire un cast entre des adresses qui n'appartiennent pas au même arbre d'héritage (le deuxième "static_cast" dans l'exemple ci-dessus). Il y a donc bien une hiérarchie verticale entre les classes que l'on "static_caster" (hors l'exemple démontre une hiérarchie plus horizontale, c'est-à-dire qu'elle est ni reliée à "Child1", "Child2" ou "Parent").

b. Dynamic_cast :

Le “dynamic_cast” est sûrement le plus intéressant de tous les “casts” à notre disposition en CPP/C++.

Ce “cast” à une première particularité très très différente des autres “casts” puisque c’est le **seul “cast” qui a lieu lors du “runtime,”** c’est-à-dire qu’il a lieu lors de l'exécution (tous les autres “casts” se font lors de la compilation, c’est le compilateur qui va faire le “cast”, le “cast” est alors statique). Attention, cela fait que le “dynamic_cast” **peut échouer** pendant l’exécution. Ainsi, il faudra mettre en forme dans du code tout **une architecture pour détecter dans le cas où le “dynamic_cast” échouerait afin de pouvoir traiter l’erreur correctement :** cette architecture s’appelle des **RTPI, “RunTime Petit Information”.** Ces RTPI sont tout un tas d’informations types qui sont stockées dans la classe mais qui nous sont cachées et qui vont nous permettre au “dynamic_cast” de vérifier si le “cast” est possible.

La deuxième particularité de ce cast, qui découle directement de la première particularité, est que **le “dynamic_cast” va fonctionner que dans le cas d’une instance polymorphique**, qu’on se trouve dans un cas de polymorphisme par sous-typage (c’est-à-dire qu’**au moins l’une des fonctions membres devra être virtuelle**, au devra donc avoir au moins une méthode dans notre classe pour pouvoir utiliser le “dynamic_cast” dessus).

Un “dynamic_cast” ne va **s'utiliser uniquement dans le cas d’une “downcast”** et vérifier si ce dernier est possible ou non. De plus, le “dynamic_cast” **va pouvoir fonctionner uniquement sur des “casts” de pointeur ou de référence :**

- Dans le cas d’une “dynamic_cast” par pointeur nous utiliserons simplement le **mot-clé “dynamic_cast”**, suivie comme d’habitude par du **type vers lequel on souhaite “caster” entre chevrons** et **l’expression à “caster” entre parenthèses.** Le “dynamic_cast” va alors nous **renvoyer**, dans le cas d’un “cast” par pointeur, **soit l’adresse qui aura été convertie soit une valeur nulle** (dans le cas ou le “cast” n’aura pas été possible).
- Dans le cas d’un “dynamic_cast” par référence, il est i**mpossible de renvoyer un pointeur nul puisque par définition la référence ne peut jamais être nulle.** Ainsi, CPP/C++ nous propose à la place de l**ever une exception qui est de type “std::bad_cast.”** Nous allons donc enfermer notre “dynamic_cast” dans un **bloc “try”** pour qui si le “cast” échoue un **bloc supplémentaire “catch”** qui récupéra notre exception.

Plus précisément, **le “dynamic_cast” va nous permettre d’ajouter des “blogins” à une application** (ex: ajouter de nouvelles librairies sous forme de “blogin” dont le code sera regrouper dans une classe). Ainsi, il faudra avoir une classe générique qui va généraliser tous nos “blogins” dont vont hériter toutes les implémentations du “blogin.” A partir de là, il sera alors possible, dans notre application principale, de chercher à “caster” le “blogin” pour pouvoir l'utiliser. Cependant, il n’est pas possible de connaître le type vers lequel on veut “caster” préalablement, il y aura qu’un pointeur vers le “blogin.” Grâce au “dynamic_cast” il est alors possible de vérifier si notre “blogin” à pour type réel le véritable “blogin” qui souhaite être utilisé.

Exemple

```
class Parent {public: virtual ~Parent(void) {} }; // Élément virtuel (RTPI)
class Child1 : public Parent {};
class Child2 : public Parent {};
class Unrelated              {};

int    main(void) {
   Child1    a; // Valeur de référence
   Parent*   b = &a; // Upcast implicite -> OK
   Child1*  c = dynamic_cast<Child*>(b); // “Dynamic_cast” par pointeur
   if (c == NULL)
      std::cout << “Conversion 1 is not OK” << std::endl;
   else
      std::cout << “Conversion 1is OK” << std::endl;
   try {
      Child2& d = dynamic_cast<Child2 &>(*b); // “Dynamic_cast” par
      std::cout << “Conversion 2 is OK” << std::endl;                référence
   }
   catch (std::bad_cast &bc) {
      std::cout << “Conversion 2 int not OK” << std::endl;
      return (1);
   }
   return (0);
}
```

Remarque

Le destructeur dans la classe “Parent” est virtuel pour bien avoir une présence d’un polymorphisme par sous-typage, d’un RTPI.

Dans le cas du premier “dynamic_cast” dans l’exemple ci-dessus, notre intense “a” est un un “Child1, l’expressions “b” à “caster” est une adresse de type “Parent” : à partir de là, vouloir faire une “dynamic_cast” d’un pointeur “Parent” vers une adresse “Child1” est possible. A l’exécution, le programme renvera “Conversion 1 is OK.”

Dans le cas du deuxième “dynamic_cast” dans l’exemple ci dessus, on cherche à caster mon “Parent” non pas vers une adresse “Child1” mais vers une référence de “Child2” (je souhaite “dynamic_caster” vers une “Child2” et non un “Child1”). Puisque, le type réel est un “Child1”, ce “dynamic_cast” échouera. Grâce au bloc “try” et “catch”, nous pouvons anticiper l’échec de “cast” en “catchant” l’exception “std::bad_cast.”

Puisqu’un “dynamic_cast” à lieu lors de l'exécution le programme compilera sans renvoyer d’erreur mais en revanche l’exécution renverra le message d’erreur du “catch” de l’exception “std::bad_cast” : l’exécution a alors échouée.

c. Reinterpret_cast :

Comme le nom l'indique, ce "cast" va nous permettre d'**effectuer des interprétations, mais également de "upcasts" et des "downcasts."** Le cas "reinterpret_cast" sera celui le plus permissif dans l'arsenal CPP/C++. Ainsi, on va pouvoir r**éinterpréter n'importe quelle adresse vers n'importe quelle autre adresse** avec bien évidemment les conséquences que cela peut avoir. Ce "cast" va être principalement utilisé pour les re-typages. pour Comme pour les précédemment types de "casts" en CPP/C++, son utilisation est la suivante :

- Un **nouveau mot-clé : reinterpret_cast** ;
- **Suivi d'un entre-chevron : <…>;**
- **Suivi d'un paramètre entre parenthèses : (...)**;

Exemple

```
int    main(void) {
   float    a = 420.042F; // Valeur de référence
   void*  b = &a; // Promotion implicite -> OK
   int*     c = reinterpret_cast<int*>(b); // Demotion explicit -> OK
   int&    d = reinterpret_cast<int &>(b); // Demotion explicit -> OK
   return (0);
}
```

Attention, dans le cas du "reinterpret_cast" aucune vérification sera effectuée. De fait, les conséquences seront de notre responsabilité.

d. Const_cast :

Parmis tous les "casts" de CPP/C++, aucun n'est **capable de faire une transformation des qualifieurs de types** : le "const_cast" va être le "cast" qui va nous permettre de réaliser cette opération. Son utilisation est la suivante :

- Un **nouveau mot-clé : const_cast** ;
- **Suivi d'un entre-chevron : <…>;**
- **Suivi d'un paramètre entre parenthèses : (...)**;

Exemple

```
int    main(void) {
   int           a = 42; // Valeur de référence
   int const* b = &a; // Promotion implicite -> OK
   int*          c = b; // Demotion implicite -> KO
   int*          d = const_cast<int *>(b); // Demotion explicite -> OK
   return (0);
}
```

Attention, **utiliser un “cosnt_cast” est un très mauvais signe** : cela veut dire que le design de code est mauvais. Il vaut mieux reprendre le code et le transformer pour éviter de l’utiliser, tiyt en gardant à l’esprit qu’il faut utiliser le plus plus de valeur constante possible (ceci afin de rendre notre programme le plus facile à maintenir et plus stable dans le temps).

27. Cast - Type Operator :

Maintenant que nous avons détaillé ensemble les différents “casts” de CPP/C++, nous allons nous intéresser à une petite notion supplémentaire : les “types cast operator.”

C’est une **nouvelle syntaxe** qui va nous permettre de **définir au sein de nos classe des opérateurs spécifiques afin de faire des conversion implicites de notre classe vers un type qui nous intéresse plus.** L’utilisation est la suivante :

- Le **mot-clé : operator;**
- Suivi d’u**n espace : “ “;**
- Puis **le type vers lequel je souhaite définir un opérateur de “cast” implicite;**

Exemple Classe

```
class Foo {
   public :
      Foo(flaot const v) : _v(v) {}
      float  getV(void)       { return (this->_v); }
      operator flaot(void)   { return (this->_v); }
      operator int(void)      { return (static_cast<int>(this->_v); }
   private :
      float                        _v;
};
```

Exemple Main	Output

int main(void) {
Foo a(420.042f):
float b = a; // Cast Implicit Foo -> flaot = operator float()
int c = a; // Cast Implicit Foo -> int = operator int()
std::cout << a.get() << std::endl;
std::cout << b << std::endl;
std::cout << c << std::endl;
}
	420.042f
	420.042f
	420

28. Cast - Explicite :

Précédemment, nous avons vu une **syntaxe récurrente pour tous les types de classe,** comme “static_cast”, “dynamic_cast”, “reinterpret_cast” et “const_cast”, à savoir : type_de_cast<type vers lequel on veut “caster”>(expression à “caster”). Cette syntaxe se rapproche de celle des appels de fonction, c’est-à-dire qu’on à directement le nom de fonction (ici c’est un mot-clé particulier et entre parenthèse une espèce de paramètre qu’on va passer et qui sera l’expression à caster).

Cette notation va nous permettre de faire un parallèle vers une autre syntaxe que nous connaissons déjà bien et qui finalement à une sémantique assez proche : soit les constructeurs. Lorsque l’on construit une instance nous allons écrire tout simplement le nom de notre instance et mettre entre parenthèses les paramètres que nous souhaitons lui envoyer quand il y en a. Finalement, on à un paramètre, une valeur, qu’on envoie à une fonction et au retour nous avons une nouvelle valeur qui a été transformée. De fait, un constructeur se rapproche d’un “cast.”

Le mot-clé “explicit” va permettre d’interdire la construction implicite d’une instance.

<table>
<tr><th>Exemple Classe</th><th>Exemple Fonction</th></tr>
<tr><td rowspan="3">class A {};
class B {};
classe C {
public:
C(A const& _) (return ;);
explicit C(B const& _) (return ;)
};</td><td>void f(C const& _) {
return ;
}</td></tr>
<tr><th>Remarque Fonction</th></tr>
<tr><td>Une fonction “f” qui prend en paramètre une référence vers une instance de classe “C” constante, définit ci-contre.</td></tr>
<tr><th colspan="2">Exemple Main</th></tr>
<tr><td colspan="2">int main(void) {
f(A()); // Conversion implicite -> OK$
f(B()): // Conversion implicite KO / Si constructeur “explicit -> OK
}</td></tr>
</table>

Remarque Classe et Main
Quand j'utilise ma fonction "f", j'appelle et je lui passe en paramètre une instance de ma classe "A." Ma fonction "f", prenant en paramètre une instance de la classe "C", il va falloir trouver un moyen de convertir ma classe "A" en une classe "C." Donc le compilateur va tester de faire une conversion implicite. Pour cela le compilateur va aller regarder ce qu'il y a à sa disposition dans la définition de notre classe "C." Nous trouvons alors, un constructeur qui nous permet de construire une instance de "C" à partir d'une classe "A."Ainsi, la première conversion implicite dans l'exemple du "main" ci-dessus va marcher puisque nous avons une conversion implicite de "A" vers "C" avant l'appel de la fonction "f." De même, la seconde conversion implicite a un comportement parfaitement équivalent : nous souhaitons appeler la fonction "f", qui prend en paramètre une instance de "C" à partir d'une instance de "B." Nous avons dans la définition de notre classe "C" un constructeur qui prend en paramètre une instance de "B." Cependant dans ce cas, nous avons rajouté le mot-clé "explicit" devant la définition du constructeur. Cela va permettre d'interdire la construction implicite d'une instance. C'est-à-dire que si nous devions avoir une classe "C" qui serait construite implicitement à partir d'une classe "A", qui est le premier exemple, alors ça fonctionne, puisque ce constructeur n'est pas noté "explicit." Cependant, si nous devions avoir une classe "C" qui serait construite implicitement à partir d'une classe "B", comme le second exemple avec la notation "explicit" cela ne fonctionnerait pas. Le compilateur ne compilera pas cette ligne respectivement parce qu'on lui indique explicitement qu'on l'on ne pas construire implicitement une classe "C" à partir d'une classe "B."

29. Cast - Conclusion :

a. Vocabulaire :

Nous avons vu la notion de :

- **Conversion** : Nous avons vu que les “casts” partaient tous d’une certaine action de “cast” particulière qui était la conversion. La conversion est ce qui va nous permettre de transformer les bits d’une valeur vers le codage d’une autre valeur, d’un autre type, afin de pouvoir effectivement convertir cette valeur.
- **Réinterprétation** : Nous avons également vu qu’il y avait un autre type de conversion particulier qui était la conversion identitaire. C’est-à-dire que les bits après conversion n’ont pas été réorganisés, on a donc exactement les mêmes bits dans le même ordre. Ce type particulier de conversion identitaire s’appelle une réinterprétation.
- **“Upcast” et “Downcat”** : Nous avons vu, ensuite, que les interprétations allaient nous permettre de pouvoir travailler avec des types d’adresses plus génériques, plus précises et d’avoir un jeu intéressant. Et qu’en particulier qu’il existait deux types particuliers de réinterprétation qui sont les “upcasts” et les “downcasts” qui vont nous permettre de naviguer dans une hiérarchie verticale de classes.
- **Type qualificatif** : Et nous avons vu également qu’il était possible de réinterpréter un type sur le qualificateur de type pour pouvoir ajouter ajouter le mot-clé “const” ou pas.

b. Les Cinq Type de Casts :

Nous avons également vu que le CPP/C++ propose cinq “casts” :

- Le “**implicit_cast**”;
- Le “**static_cast**”;
- Le “**dynamic_cast**”;
- Le “**reinterpret_cast**”;
- Et le “**const_cast**”.

Cast	Conv	Reinter.	Upcast	Downcast	Type qual.
Implicit	YES		YES		
Static_cast	YES		YES	YES	
Dynamic_cast			YES	YES	YES
Const_cast					
Reinterpret_cast		YES	YES	YES	YES
Legally C cast	YES	YES	YES	YES	YES

Cast	Sematic Check	Reliable On Run	Tested At Run
Implicit Static_cast Dynamic_cast Const_cast Reinterpret_cast	YES YES YES	YES YES	YES
Legally C cast			

30. Templates :

Les templates, CPP/C++, vont nous permettre d'écrire des programmes une précision absolument bluffante. Plus précisément, les templates sont une syntaxe spéciale au CPP/C++ qui vont nous permettre d'écrire des patrons de morceau de code, des "templates" (c'est-à-dire du code à trou). À partir de cela, le compilateur utilisera ces templates pour gérer le code à notre place.

a. Introduction :

Les templates vont nous permettre, grâce à leurs syntaxes spécifiques, de **définir du code dans lequel on va introduire des variables de types**. Plus spécifiquement, ces templates vont nous permettre de définir des fonctions templates et des classes templates. **Ces patrons de code, ces exemples, ces modèles de code à trou pourront ensuite être utilisés par notre compilateur pour instancier ces fameux "trous", là où il y a les variables de types, avec les types qui nous intéressent vraiment et les utiliser au bon endroit dans votre code ensuite.** Par convention, les fichier dans lesquelles se trouvent des templates prendre le suffixe **".tpp"**;

b. Syntaxe :

Pour indiquer à mon compilateur qu'il va s'agir d'un template de fonction je doit ajouter les choses suivantes à la fonction en question : **template< typename … >**

- Le mot-clé "**template**" qui va nous permettre d'indiquer qu'il s'agit d'un template de fonction.
- Le mot-clé "**typename**", entre-chevron, suivi du nom de la nouvelle variable de type, qui va me permettre d'introduire la variable en question.

Pour **utiliser le nouveau template** il va falloir d**emander au compilateur de nous faire un instanciation de ce dernier.** CPP/C++ offre deux moyens d'effectuer cette instanciation de templat**e** :

- Soit une **instanciation explicite** :
  appeler_la_fonction<type_vers_lequel_on_veut_instancier_le_template>(paramètres);
- Soit une **instanciation implicite** :
  appeler_la_fonction(paramètres);

Les templates vont également nous permettre d'écrire des **fonctions templates** mais également **des classes et des structures templates**. La syntaxe et la logique restent les mêmes.

Les templates, de classe ou de fonction, nous permettent d'obtenir une version générique de notre classe ou de notre fonction. Le template en question est instanciée avec un ou plusieurs types qu'on allait donner en paramètre.

Exemple 1

```
template< typename T >
T const&    max(T const& a,T const& y) { // Const car “x” et “y” restent
   return ( x >= y ? x : y):                inchangées pendant l’appel de la fonction
}

int    foo(int x) {
   std::cout << “Long computing time” << std::cout;
   return (x);
}

int    main(void) {
   int      a = 23;
   int      b = 42;
   float    c = -2.7f;
   float    d = 4, 2f;

   std::cout << “Max of “ << a << “ and “ << b << “ is “;
   std::cout << max<int>(a, b) << std::endl; // Interprétation explicite
   std::cout << “Max of “ << a << “ and “ << b << “ is “; // Interprétation implicite
   std::cout << max(a, b) << std::endl;
   std::cout << “Max of “ << c << “ and “ << d << “ is “;
   std::cout << max<float>(c, d) << std::endl; // Interprétation explicite
   std::cout << “Max of “ << c << “ and “ << d << “ is “; // Interprétation implicite
   std::cout << max(c, d) << std::endl;

   int    ret = max<int>(foo(a), foo(b));
   std::cout << “Max of  “ << a <<  “ and “ << b << “ is “;
   std::cout << ret << std::endl;
}
```

Remarque

La fonction “max” va nous permettre de comparer deux types “T” entre eux est retourner le plus grand. Ici, il est important d’utiliser des références sur la variable de types “T” pour nous permettre d’éviter un surcharge de mémoire si c’est une grosse classe, qui est volumineuse.
Dans la deuxième utilisation de la fonction “max” pour “a” et “b” dans l’exemple ci-dessus ou se rend compte qu’il n’est pas forcément nécessaire d'indiquer le type vers lequel on veut instancier notre template. Le compilateur le comprend seul, c’est une interprétation implicite et non explicite. Attention, dans des cas d’instanciation de templates plus important, l'instanciation implicite est à éviter.

c. Type par Défaut :

Il est possible de **passer des valeurs par défaut à nos variables de type de nos templates.** C'est-à-dire que si aucune type est passée à ces dernières, elles prendront le type par défaut. **Syntaxe = template< typename name = type_par_défaut >**

Exemple 1
template< typename T = float >

d. Spécialisation de Template :

Les **templates de classe ou de notre fonction**, instanciés avec un ou plusieurs types qu'on allait donner en paramètre, **partagent tous un code rigoureusement identique qui est directement issu du template**. De fait, **afin de différencier légèrement ces codes il faudra utiliser une nouvelle syntaxe : celle de la spécialisation de template.** Pour spécialiser une classe il faut rajouter la syntaxe suivant à la déclaration de classe: **class Name<types> et la classe doit contenir un constructeur par défaut private.**

Il peut y avoir des **spécialisations partielles ou complètes suivant le cas.** Ainsi, il est possible de spécialiser partiellement et complètement une classe template. Toutefois, il n'est **pas possible de faire spécialisation partielle d'une classe template**. Puisqu'une spécialisation partiellement d'une revient à faire une surcharge. Une spécialisation est complète quand tous ces types sont par défaut. En revanche, une spécialisation est partielle si certains de ces types ne sont pas par défaut.

Exemple Pas de Spécialisation

```
template< typename U, typename T >
class   Pair< T, U > { //Dans le cas ou aucun des paramètres doit être spécialisés
   public:
      Pair< T, U >(int lhs, U const& rhs): _lhs(lhs), _rhs(ths) {}
      ~Pair< T, U >(void) {};
      int           getLhs(void) const { return (this->_lhs); }
      U const& getRhs(void) const { return (this->_rhs); }
   private:
      int           _lhs;
      U const& _rhs;
      Pair< T, U >(void);
};
```

Exemple Spécialisation Partielle

```
template< typename U >
class    Pair< int, U > { // Dans le cas où le premier paramètre sera toujours int
   public:
      Pair< int, U >(int lhs, U const& rhs): _lhs(lhs), _rhs(ths) {}
      ~Pair< int, U >(void) {};
      int            getLhs(void) const { return (this->_lhs); }
      U const& getRhs(void) const { return (this->_rhs); }
   private:
      int            _lhs;
      U const& _rhs;
      Pair< int, U > (void);
};
```

Exemple Spécialisation Partielle

```
template< >
class    Pair< bool, bool > { //  Dans le cas où les deux paramètre sont bool bool
   public:
      Pair< bool, bool >(bool lhs, bool rhs) { etc… }
      ~Pair< bool, bool >void) {};
      bool   getN(void) const { return (this->_n); }
   private:
      int      _n
      Pair< bool, bool >(void);
};
```

e. Conclusion :

Les templates sont l'élément de syntaxe le plus puissant de CPP/C++. Les templates permettent également d'utiliser la bibliothèque standard de CPP/C++ qu'on appelle la "STL", pour "Standard Templates Library."

31. Standard Templates Librairy :

La **STL**, **"Standard Templates Library",** contient deux choses très importantes :

- Les **conteneurs**, qui sont des **conteneurs génériques templater qui peuvent contenir n'importe quoi et qui sont d'implémentation très bonnes.**
- Et les **algorithmes qui permettent d'opérer des opérations pré-implémenter sur une collection** et qui peut être un conteneur standard de la STL ou bien un conteneur que nous avons créé.

Exemple Conteneur

```
#include <iostram>
#include <map> // std::map
#include <vector> // std::vector
#include <list> // std::list

class IOperation;

int    main(void) {
   std::list<int>                          lst1; // Conteneur de liste chaînée template en Int
   std::map<std::string, IOperation*> map1;
   std::vector<int>                        v1;
   std::vector<int>                        v2(42, 100);
   std::list<int>::const_iterator          it; // Équivalent conteneur STL d'un "pointeur"
   std::list<int>::const_iterator          ite = lst1.end(); // Dernière nodes de lst1;

   lst1.push_back(1);
   lst1.push_back(17);
   lst1.push_back(42);
   map1["opA"] = new OperationA;
   map1["opB"] = new OperationB;
   for (it = list1.begin(); it != ite; ++it
      std::cout << *it << std::endl;
   return (0);
}
```

Exemple Algorithmes Pré-implémenter

```
void    displayInt(int i) {
   std::cout << i << std::endl;
}

int      main(void) {
   std::list<int>    lst;

   lst.push_back(23);
   lst.push_back(12);
   lst.push_back(20);
   lst.push_back(18);
   for_each(lst.begin(), lst.end(), displayInt);
   return (0);
}
```