ORSYS formation

Learning for success

VOTRE SUPPORT DE COURS

FORMATION EN CLASSE A DISTANCE

Programmation C++, perfectionnement

25 - 28 Juin 2024

Bienvenue **chez ORSYS**

CE QUE VOUS DEVEZ SAVOIR

Vos deux espaces en ligne accessibles depuis un ordinateur, une tablette, un smartphone

https://docadmin.orsys.fr

- Déclarer votre présence en cours (vous devez signer chaque demi-journée).
- Valider vos acquis.
- Évaluer la formation.

Le mot de passe pour accéder à cet espace vous sera donné par le formateur.

L'évaluation est à compléter juste avant la fin de la session.

La validation des acquis est à remplir en 2 étapes, au début et à la fin de la session.

https://myorsys.orsys.fr

- Récupérer la version digitale de votre support de cours et des autres ressources pédagogiques.
- Retrouver l'historique de vos formations.
- Garder contact avec vos formateurs et les autres participants.

INFORMATIONS GÉNÉRALES

Horaires

Les sessions commencent à 9h15 le premier jour et à 9h les jours suivants. Elles se terminent à 17h30 sauf le dernier jour où elles prennent fin à 17h (excepté pour les sessions de quatre ou cinq jours où elles se terminent lors de la pause de l'après-midi).

Pour les formations en présentiel

- Nos centres de formation ouvrent à partir de 8h45.
- Les déjeuners et pauses-café vous sont offerts.
- Vous trouverez les plans d'évacuation affichés dans chaque centre.

L'accessibilité

Que vous suiviez une formation en présentiel ou à distance, n'hésitez pas à contacter notre référent handicap : psh-acceuil@orsys.fr

L'environnement

Les éco-gestes en présentiel et en distanciel.

Éteindre l'ensemble des périphériques

Utiliser nos poubelles de tri sélectif

Éteindre les lumières lors de votre départ

Garder votre gobelet pour vous hydrater

BONNE FORMATION !

Osez explorer le champ des possibles avec ORSYS FORMATION

Plus de 2000 formations dans les domaines du management, développement personnel, technologies numériques et principaux métiers de l'entreprise.

Retrouvez tous nos programmes de formation sur www.orsys.com

L'ADN ORSYS

Notre mission

Accompagner les individus et les organisations dans leur montée en compétence.

Nos valeurs

- LA QUALITÉ DE SERVICE
- L'ENGAGEMENT ET LA PERFORMANCE
- LA BIENVEILLANCE ET LA RECONNAISSANCE

160 000 PERSONNES FORMÉES

96,5 % DE TAUX DE SATISFACTION

22 000 SESSIONS ORGANISÉES

47 ANS D'EXPÉRIENCE

UN RÉSEAU DE **2 200** FORMATEURS EXPERTS DE TERRAIN

DONNÉES 2022

3 critères qualité **qui font la différence**

1. La qualité de l'offre

→ UNE OFFRE RICHE ET MULTIMODALE

40 domaines d'expertise, plus de **2 000** formations proposées en inter et intra, **600** parcours digitaux et des solutions à la carte.

→ DES INTERVENANTS EXPERTS

Professionnels de terrain, nos formateurs transmettent leur expérience métier aux participants.
Formés à notre pédagogie, ils sont rigoureusement validés par nos équipes.

→ UNE PÉDAGOGIE IMMERSIVE ET INNOVANTE

Nos ingénieurs pédagogiques et formateurs élaborent des méthodes spécifiques pour favoriser la transmission des connaissances en privilégiant la mise en pratique directe.

2. La qualité de service

→ UN INTERLOCUTEUR DÉDIÉ

Un consultant formation proche de chez vous et dédié à votre entreprise est là pour vous accompagner : étude de vos besoins, proposition de solutions adaptées en termes de pédagogie, d'organisation et de financement.

→ UN ENVIRONNEMENT OPTIMAL EN PRÉSENTIEL ET À DISTANCE

+ de 35 centres en France, en Belgique, en Suisse et au Luxembourg

- Facilités d'accès et salles de cours spacieuses, lumineuses et adaptées aux personnes en situation de handicap.
- Pour le distanciel, un accès à votre environnement adapté, simple et fiable : une connexion Internet suffit !

3. La qualité de notre organisation

→ LA QUALITÉ DE NOS PROCESSUS

ORSYS est certifié par des référentiels officiels nationaux pour la qualité de ses processus.

→ NOS ENGAGEMENTS

→ NOS CHARTES ÉTHIQUES

ORSYS formation

Learning for success

VOTRE SUPPORT DE COURS

FORMATIONS ORSYS

Toutes nos solutions

ORSYS *vous accompagne dans le développement des compétences*

→ L'INTER → L'INTRA → LE SUR-MESURE

ORSYS formation

Formation C++ : perfectionnement

PAR ALAIN CARIOU, JUIN 2024

Sommaire

- Quelques rappels sur le C++
- Les nouveautés de C++ 11
- La gestion des opérateurs
- Conversion et RTTI
- La généricité
- La Standard Template Library
- Les nouveautés de la librairie standard
- Boost et ses principes
- Utilisation avancée de l'héritage

Prologue

Objectifs pédagogiques

Maîtriser la mémoire, des pointeurs et des références

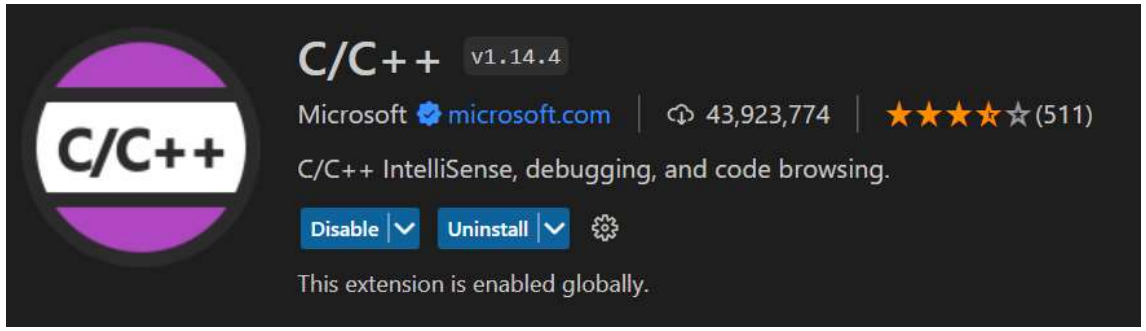
Implémenter la généricité en C++

Découvrir la bibliothèque standard STL

Utiliser les apports de la norme C++11

Mettre en place son environnement - 1

- Pour développer en C++ sur Visual Studio Code, vous aurez besoin d'installer l'extension C/C++.



- Vous aurez aussi besoin d'installer un compilateur C++. Il en existe plusieurs sous Windows.

Mettre en place son environnement - 2

- Vous pouvez par exemple télécharger Mingw-w64 sur le site suivant :
    - *https://www.msys2.org/*
- Après l'installation, un terminal devrait s'ouvrir. Exécutez la commande suivante pour installer les outils nécessaires :
    - ***pacman -S --needed base-devel mingw-w64-x86_64-toolchain***
- Enfin ajoutez le répertoire ***\mingw64\bin*** aux variables d'environnement de votre **Path**.
- Pour vérifier que tout fonctionne bien, lancez la commande suivante dans un terminal :
    - ***g++ --version***

I – Rappel

Construction, initialisation et embarquement d'objets

- L'**embarquement d'objet** permet à une classe dérivée d'être attribuée à sa classe parente.

- Attention cela cause la perte des données spécifiques à la classe dérivée.

- L'utilisation de pointeurs ou référence permet d'éviter l'embarquement d'objet, et donc la perte de données.

```cpp
class ParentClass {
public:
    int data;
    ParentClass(int d) : data(d) { }
    virtual void display() {
        std::cout << "Parent class : " << data << std::endl;
    }
};

class ChildClass : public ParentClass {
public:
    int value;
    ChildClass(int x, int y) : ParentClass(x), value(y) { }
    void display() override {
        std::cout << "Derived class : " << data << " - "
        << value << std::endl;
    }
};

int main() {
    ChildClass child(10, 200);
    ParentClass parent = child;
    parent.display();
    return (0);
}
```

Les classes d'allocation mémoire

- C++ est un langage qui permet à l'utilisateur de **manipuler la mémoire** de manière dynamique.

- Bien que l'on puisse utiliser ***malloc()*** ou ***calloc()***, il est préférable d'utiliser les opérateurs ***new*** et ***delete***. Ce qui implique une allocation et une libération de la mémoire manuelle !

Cela nécessite de faire attention aux fuites mémoires qui peuvent occasionner des ralentissements, crashs et corruptions de données !

La pile et le tas

- La mémoire utilisée par un programme C++ est divisée en plusieurs parties dont les deux principales sont la **pile** (**stack**) et le **tas** (**heap**).

La pile	Le tas
Est utilisée pour l'allocation de mémoire statique	Est utilisé pour l'allocation de mémoire dynamique
Stocke les variables locales	Stocke les variables globales
L'allocation et la désallocation sont gérées par le compilateur.	L'allocation et la désallocation sont gérées manuellement par le développeur.
A un temps d'accès très rapide	A un temps d'accès « lent »
Structure de données linéaire	Structure de données hiérarchique

L’opérateur new

- L’opérateur ***new*** permet **d’allouer de la mémoire** de manière dynamique. Si suffisamment d’espace est disponible, la mémoire est allouée et un pointeur sur la zone mémoire est retournée. En cas d’échec, une exception **bad_alloc** est levée.

- Plusieurs syntaxes sont possibles :
  - ***pointeur = new type;***
  - ***pointeur = new type(valeur);***
  - ***pointeur = new type[taille];***

L’opérateur delete

- L’opérateur **delete** permet de libérer la mémoire et appelle également le destructeur de classe si possible.

- Deux syntaxes sont possibles en fonction du contexte :
  - ***delete pointeur;***
  - ***delete[] pointeur;***

- C’est très important ! Si vous ne libérez pas la mémoire allouée, votre programme risque d’avoir des fuites mémoires !

Les opérateurs new et delete : exemple

- Exemple :

```
int main () {
    int *ptr1 = NULL;
    ptr1 = new int;
    *ptr1 = 7;
    double *ptr2 = new double(21.42);
    int *ptr3 = new int[4];

    std::cout << "Valeur du pointeur 1 : " << *ptr1 << std::endl;
    std::cout << "Valeur du pointeur 2 : " << *ptr2 << std::endl;

    delete ptr1;
    delete ptr2;
    delete[] ptr3;
    return 0;
}
```

Les fuites mémoires

- Comme vu plus haut, une mauvaise gestion de la mémoire peut entraîner des **fuites mémoires**. La mémoire de la machine peut se trouver surchargée, causant des **ralentissements** voir même un **crash**.

- **Différents outils** permettent de détecter les fuites mémoires comme le fameux **Valgrind** sous Linux, **WinDBG** ou **Deleaker**.

- L'utilisation de **bonnes pratiques de développement**, **d'outils modernes**, et de **types de données appropriés** (pointeurs intelligents, conteneurs standards, etc) permettent de limiter les fuites.

Les types constants

- Il est possible de déclarer une **variable** en tant que **constante** en utilisant le mot-clé « ***const*** » :

```
const int i = 10;
```

- La valeur de la variable ne pourra plus être modifiée ensuite, sinon cela créerait une erreur de compilation.

- Il est aussi possible de déclarer des **méthodes** en **constante** : cela signifie que la méthode ne peut pas modifier les champs non statiques ou appeler des méthodes qui ne sont pas constantes.

```
int getAge() const {
    return age;
}
```

Le mot-clé mutable

- Le mot-clé **mutable** permet d'autoriser la modification d'une variable membre à l'intérieur d'une méthode constante. Attention à n'utiliser cela que lorsque c'est strictement nécessaire !

- Exemple :

```
class MyClass {

private:
    mutable int mutableVar;

public:
    int getMutableVar() const {
        this->mutableVar = 42;
        return mutableVar;
    }
};
```

Lazy computation

- Le mécanisme de « **lazy computation** », aussi appelé « **lazy evaluation** » ou « **évaluation différée** » consiste à effectuer le calcul d'une valeur uniquement lorsqu'on souhaite accéder à celle-ci.

- Cela peut permettre d'**optimiser les performances** d'un programme en évitant des calculs inutiles.

- Ce mécanisme est souvent effectuée lors de **surcharges d'opérateurs**, à travers des **lambdas** ou bien encore dans des **foncteurs**.

Le contrôle d'accès

- Les propriétés et méthodes d'une classe peuvent avoir différents niveaux d'accessibilité pour accéder à leurs données :
    - **public** : les membres de la classe sont accessibles par n'importe quelle fonction.
    - **private** : les membres de la classe ne sont accessibles qu'à l'intérieur de la classe et par les « *amis* » de la classe.
    - **protected** : les membres de la classe sont considérés comme privés mais sont aussi accessibles aux classes dérivées de cette classe.
- L'accès par défaut des membres d'une classe est « ***private*** ».

Amitié : les méthodes « friend »

- Une méthode « ***friend*** » est une méthode définie en dehors d'une classe mais qui a accès à tous les membres protégés et privés de cette classe.

- Une méthode « *amie* » est déclarée dans la classe avec le mot-clé « ***friend*** » :
  - ***friend int calcArea();***

- A noter que les méthodes « ***friend*** » n'ont pas accès au pointeur « ***this*** » ! Enfin il faut considérer l'amitié en C++ comme une extension à **l'encapsulation**.

Amitié : les classes « friend »

- Une classe peut aussi être déclarée en tant que « ***friend*** » au sein d'une autre classe. Cela lui permet d'avoir, là aussi, accès aux membres protégés et privés de cette classe.

- Deux syntaxes sont possibles :

```
class FriendClass {};

class MyClass {
    friend FriendClass;
};
```

```
// Syntaxe à privilégier
// si FriendClass n'a pas encore été déclarée
class MyClass {
    friend class FriendClass;
};
```

Le mot-clé « explicit »

- Avec C++ 11, les constructeurs deviennent des **constructeurs de conversion**. C'est à dire que le compilateur peut convertir un objet en un autre type suivant ses besoins.

- Le mot-clé **« explicit »** permet d'indiquer au compilateur qu'un constructeur ne doit pas être utilisé pour des **conversions implicites**.

- Par exemple une fonction qui prendrait en paramètre un objet pourrait le convertir implicitement en un *int*, entraînant alors des comportements inattendus.

```
class MyClass {
public:
    int value;
    explicit MyClass(int x) : value(x) { }
};
```

Destructeur virtuel

- Si on essaye de supprimer un objet d'une classe dérivée à travers un pointeur, on peut obtenir des **comportements inattendus**.

- L'utilisation d'un **destructeur virtuel** permet de garantir une suppression propre de l'objet de la classe dérivée.

- Enfin les destructeurs virtuels sont aussi utilisés avec les **interfaces** et les **classes abstraites pures**.

Destructeur virtuel : exemple

```cpp
class Base {
  public:
    Base() { std::cout << "Constructeur base" << std::endl; }
    virtual ~Base() { std::cout << "Destructeur base" << std::endl; }
};

class Derived: public Base {
  public:
    Derived() { std::cout << "Constructeur derived" << std::endl; }
    ~Derived() { std::cout << "Destructeur derived" << std::endl; }
};

int main() {
  Derived *d = new Derived();
  Base *b = d;
  delete b;
  return 0;
}
```

Stratégie de gestion des exceptions

- En C++ les erreurs peuvent être gérées grâce aux **exceptions**. Il s'agit de l'interruption du programme lorsqu'une erreur est rencontrée, ce qui permet de traiter ainsi ces erreurs.

- L'instruction « ***try*** » permet de **tester un bloc de code** où une exception pourrait être levée.
- L'instruction « ***catch*** » permet de **définir un bloc de code à exécuter** lorsqu'une exception est rencontrée.
- L'instruction « **throw** » permet de **lever directement une exception** en cas de problème et de personnaliser les erreurs.

La génération des exceptions : exemple

- Ici on peut voir la gestion de plusieurs exceptions en fonction de leur type.

- La syntaxe **(...)** signifie que ce bloc traitera toutes les **exceptions standards du C++**.

```cpp
int num1 = 10;
int num2 = 0;
int result;

try {
    if (num2 == 0) {
        throw num2;
    }
    result = num1 / num2;
    std::cout << result << std::endl;
} catch (int number) {
    std::cerr << "Error : division by " << number << std::endl;
} catch (std::exception& e) {
    std::cerr << "Error : an exception occured !" << e.what() << std::endl;
} catch (...) {
    std::cerr << "An default exception occured !" << std::endl;
}
```

La construction d'une hiérarchie d'exception - 1

- C++ dispose d'une liste d'exceptions standards qui héritent toutes de la classe ***std::exception*** définie dans ***<exception>*** afin de gérer des cas d'erreurs plus spécifiques.

- La classe ***std::exception*** dispose aussi d'une méthode ***what()*** qui retourne une chaîne de caractères donnant des informations sur l'exception levée.

- Il est aussi possible **d'implémenter vos propres exceptions** en héritant d'une de ces classes.

La construction d'une hiérarchie d'exception - 2

- Tableau des différents types d'exceptions héritant de **std::exception**.

Exception	Description
std::exception	La classe parente de toutes les exceptions du C++
std::bad_alloc	Une exception levée par ***new***
std::bad_cast	Une exception levée par ***dynamic_cast***
std::bad_exception	Permet de gérer les exceptions inattendues
std::bad_typeid	Une exception levée par ***typeid***
std::logic_error	Une exception pouvant être détectée à la compilation. Plusieurs exceptions héritent de celle-ci.
std::runtime_error	Une exception qui ne peut pas être détectée à la compilation. Plusieurs exceptions héritent de celle-ci.

L’utilisation des exceptions

- On pourrait considérer les mécanismes de gestion des exceptions comme une structure de contrôle supplémentaire dans un programme, or cela doit être évité. Car **la gestion des exceptions est la gestion des erreurs** ! Les exceptions sont optimisées pour cela.

- Ainsi le code est séparé en deux catégories : le **code classique où l’erreur est détectée** et le **code traitant les erreurs**.

- N’oubliez pas de **libérer les ressources** utilisées par le programme lorsqu’une exception survient sur celles-ci : fermer un fichier, libérer la mémoire, déverrouiller un mutex, etc.

Les espaces de nommage

- Un **espace de nommage**, aussi appelé **namespace**, est un ensemble regroupant différentes variables, fonctions, classes, …, au sein d'un même espace.

- Cela permet de différencier par exemple des fonctions qui auraient des noms similaires dans différentes librairies.

- On déclare un espace de nommage grâce à l'instruction « ***namespace*** ».
- On peut indiquer que l'on utilise un espace de nommage particulier grâce au mot-clé « ***using*** ».

Les espaces de nommage : exemple

- Voici un exemple montrant l'utilisation de deux **namespaces** *foo* et *bar* :

- On notera l'utilisation du namespace ***foo*** à travers l'instruction ***using namespace foo***.

- Cela permet l'appel de la fonction ***my_function()*** sans avoir à répéter le nom de l'espace de nommage.

```cpp
#include <iostream>

namespace foo {
    void my_function() {
        std::cout << "Inside foo namespace" << std::endl;
    }
}

namespace bar {
    void my_function() {
        std::cout << "Inside bar namespace" << std::endl;
    }
}

using namespace foo;

int main() {
    my_function();
    bar::my_function();
    return 0;
}
```

Points essentiels à retenir

- La gestion de la mémoire est extrêmement importante ! Toute **mémoire allouée** doit être **libérée** !
- Différents **opérateurs** existent et peuvent être **surchargés** en fonction des besoins.
- Des types de données peuvent être **constants** mais sujet à des modifications grâce au mot-clé **mutable**.
- Le contrôle d'accès (**public**, **protected**, **private**, **friend**) permet de protéger les données d'une classe.
- Le code d'un programme peut être séparé en deux parties avec les **exceptions** : le code de **détection** des erreurs et le code de **gestion** des erreurs.
- L'utilisation des **namespaces** permet de mieux séparer le code.

II – Utilisation avancée de l’héritage

Héritage versus embarquement

Avantages de l'héritage	Avantages de l'embarquement (composition)
- Réduction du code à écrire - Réutilisation du code - Héritage des propriétés et méthodes de la classe parente - Polymorphisme	- Plus de flexibilité que l'héritage : la classe n'est pas liée à une classe parente - Meilleur contrôle sur les membres de la classe - Favorise l'encapsulation - Évite les problèmes de l'héritage (couplage fort, héritage en diamant) - Facilité de maintenance

Souvent, un mélange des deux est à privilégier.

Héritage privé et héritage protégé

- En fonction du type d'héritage, l'accès aux données change selon le tableau suivant :

	Héritage public	Héritage protected	Héritage private
Accès public	public	protected	private
Accès protected	protected	protected	private
Accès private	interdit	interdit	interdit

- Ainsi les données publiques d'une classe parente deviennent publiques, protégées, ou privées en fonction de l'héritage de la classe dérivée. Les données protégées deviennent soit protégées, soit privées. Et les données privées de la classe parente sont toujours inaccessibles.

Exportation des membres cachés avec la clause « using »

- Une des utilisations de la clause « **using** » consiste à permettre l'exportation de membres cachés d'une classe de base vers une classe dérivée.

- Les membres exportés **deviennent ainsi des membres de la classe dérivée**. Cependant cela peut permettre de rendre publique des membres protégés, **réduisant aussi l'encapsulation** …

```
class Base {
protected:
    int number;
};

class Derived : public Base {
public:
    using Base::number;
};

int main() {
    Derived myClass;

    myClass.number = 721;
    std::cout << myClass.number << std::endl;
    return 0;
}
```

Héritage multiple

- Le C++ supporte **l'héritage multiple**. Cependant lorsque deux classes ont des membres au nom similaire, cela pose problème et empêche la compilation.

- Pour éviter cela, on doit spécifier à partir de quelle classe parente la méthode ou la variable est appelée :

- ***derivedClass.BaseClass::function();***

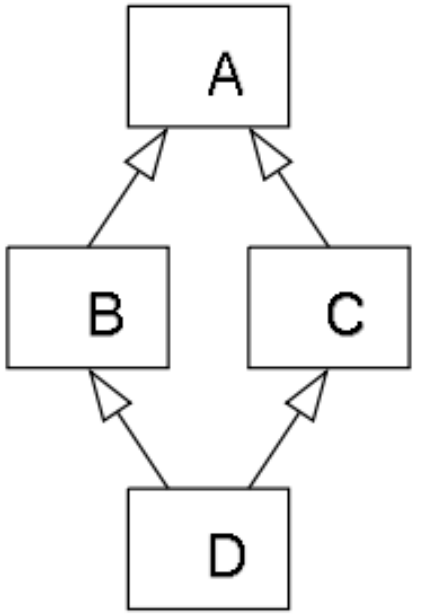
```cpp
class A {
public:
    A() { std::cout << "Constructeur de A" << std::endl; }
    void f() { std::cout << "Fonction A" << std::endl; }
};

class B {
public:
    B() { std::cout << "Constructeur de B" << std::endl; }
    void f() { std::cout << "Fonction B" << std::endl; }
};

class C: public A, public B {
public:
    C() { std::cout << "Constructeur de C" << std::endl; }
};

int main() {
    C myClass;

    myClass.f();  // Ne compile pas
    myClass.A::f(); // Compile
    return 0;
}
```

L’héritage en diamant

- Quand on parle d’héritage multiple, on aborde souvent le problème de l’**héritage en diamant**.

- C’est à dire qu’une classe va hériter de plusieurs classes qui elles-mêmes héritent d’une même classe.

- Ainsi on peut se retrouver avec plusieurs appels du constructeur de la classe originelle, et donc plusieurs instances créées sans le vouloir !



L’héritage en diamant : exemple

- Ici, les classes B et C dérivent de A.
- Et la classe D dérive de B et C.
- Pour éviter le problème de l’héritage en diamant, on spécifie alors un **héritage virtuel** grâce au mot-clé « ***virtual*** ».

```
class A {
public:
    A() { std::cout << "Constructeur de A" << std::endl; }
};

class B: virtual public A {
public:
    B() { std::cout << "Constructeur de B" << std::endl; }
};

class C: virtual public A {
public:
    C() { std::cout << "Constructeur de C" << std::endl; }
};

class D: public B, public C {
public:
    D() { std::cout << "Constructeur de D" << std::endl; }
};
```

Principes de conception

- **L'héritage** peut amener son **lot de problématiques** pouvant grandement **complexifier le code** si on n'y fait pas attention.

- De cela découle différents principes de conceptions comme **la substitution de Liskov** ou bien encore **l'inversion des dépendances**.

Principes de conception : la substitution de Liskov - 1

- La **substitution de Liskov** est formulée comme suit :

« *Si q(x) est une propriété démontrable pour tout objet x de type T, alors q(y) est vraie pour tout objet y de type S tel que S est un sous-type de T.* »

- Cela signifie que les **classes dérivées puissent être utilisées à la place des classes de base** sans pouvoir provoquer de comportements imprévus ou d'erreurs dans le programme.

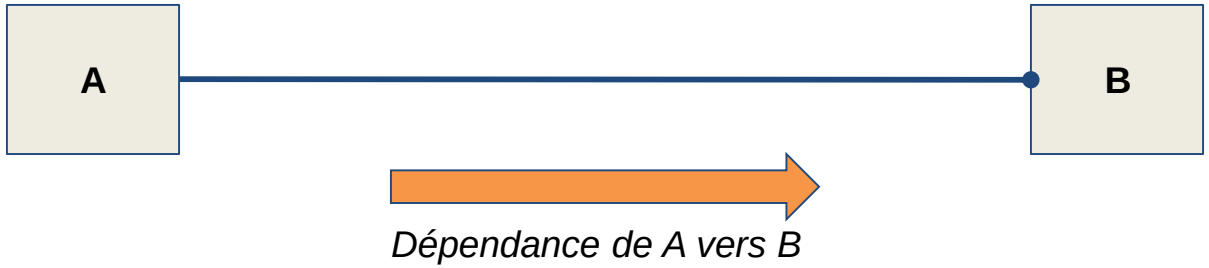
- Cela permet aussi de s'assurer que le code est **robuste** et **flexible**.

Principes de conception : la substitution de Liskov - 2

- On peut garantir la substitution de Liskov en respectant les règles suivantes :
- Les classes dérivées doivent avoir les **mêmes signatures de méthode** que les classes de base.
- Si la classe de base spécifie qu'une condition doit être remplie avant l'appel d'une méthode, la **classe dérivée ne peut pas exiger une condition plus stricte**.
- Si la classe de base spécifie qu'une condition doit être vraie après l'exécution d'une méthode, la **classe dérivée doit garantir que cette condition est toujours vraie**.
- Les **propriétés** qui doivent être vraies tout au long de la durée de vie de la classe de base **doivent également être respectés** par la classe dérivée.

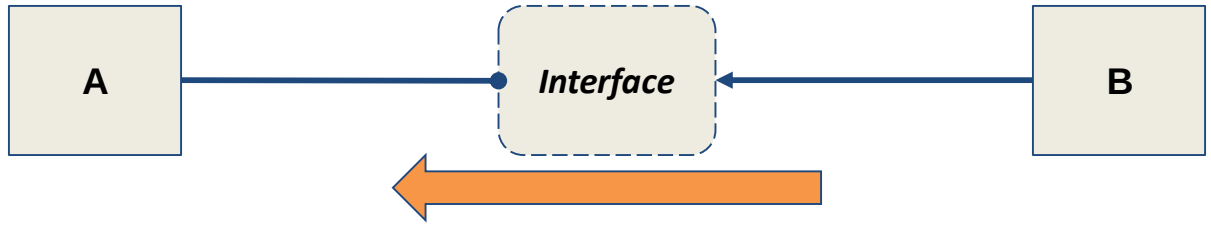
Principes de conception : l'inversion de dépendances - 1

- Imaginons deux modules A et B contenant une fonction x du Module A qui appelle une fonction y du Module B. Cela veut dire qu'il y a une dépendance du module A au Module B.
- A la fois au niveau de l'exécution où A dépend de B, mais aussi au niveau du code source où le module A contient des déclarations comme « *include* », « *using* », … , qui réfèrent au Module B.
- Les deux modules ne peuvent pas être déployés séparément et un changement dans le Module B implique alors forcément la recompilation et le redéploiement du Module A.



Principes de conception : l'inversion de dépendances - 2

- On peut cependant inverser la dépendance du code source sans changer la dépendance d'exécution : en supprimant la dépendance originale, en ajoute une interface et en faisant que A dépende de l'interface, et que B dérive de l'interface.
- Cela change le sens de la dépendance de code source pour pointer maintenant de B vers A. Maintenant, la dépendance de B pointe dans le sens inverse de la dépendance d'exécution.
- Ce qui permet finalement aux modules A et B d'être déployés séparément.



Règle d'implémentation des interfaces

- Une **interface** décrit le comportement d'une classe sans spécifier d'implémentation particulière de cette classe. Il s'agit ici d'une **classe abstraite pure** !

- En C++, les interfaces sont donc implémentées à travers les **classes abstraites**.

- Pour des questions de gestion de mémoire, il faut toujours utiliser un destructeur virtuel vide ou non-virtuel « ***protected*** » :

- Exemple :

```
virtual ~IShape() {};
```

Points essentiels à retenir

- L'héritage peut devenir assez complexe suivant la manière dont il est mis en place.
- La **gestion des accès** change complètement en fonction du **niveau de protection** de l'héritage.
- Des membres protégés peuvent être exportés avec la clause « **using** »
- Dans le cas d'un héritage multiple ou plusieurs membres possèdent le même nom, la classe enfant doit **appeler directement le membre en indiquant son parent**.
- Dans le cas d'un héritage en diamant, il faut spécifier qu'il s'agit d'un héritage **virtuel**.
- Différents **principes de conception** peuvent être appliqués pour avoir un héritage plus lisible et plus maintenable.

III – Les nouveautés de C++ 11

nullptr

- A partir de C++11 apparaît le pointeur **nullptr**. Il s'agit d'un pointeur ayant le type ***std::nullptr_t***.

- Il peut être implicitement converti en n'importe quel type de pointeur mais pas en d'autre type de données. Un **pointeur null** pouvant être égal à **0**, cela permet d'éviter une conversion en int.

- Cela permet **d'améliorer la lisibilité de code écrit et la maintenance** des bibliothèques C++.

```
int *ptr_x = nullptr;
```

La directive delete

- La directive **delete** permet d'indiquer au compilateur qu'une fonction ne doit pas être générée automatiquement.

- Cela permet d'éviter des conversions de type implicite.

- Dans l'exemple ci-contre, on s'assure que la méthode *calc* ne prenne en paramètre que des int :

```
int calc(int nb);
int calc(float nb) = delete;
int calc(double nb) = delete;
```

La directive default

- La directive **default** permet d'indiquer au compilateur de générer automatiquement le constructeur par défaut, par copie, de mouvement ou le destructeur approprié.

- Cela permet de s'assurer que la fonction soit bien générée par défaut et d'avoir un code plus lisible.

```
class MyClass {
    public:
     MyClass() = default;
     MyClass(std::string something);
};
```

La délégation de constructeurs

- La **délégation de constructeurs** est un mécanisme qui va permettre de **diminuer la répétitivité** du code en appelant un autre constructeur dans un constructeur.

- Cela se traduit à travers la syntaxe suivante :
    - ***constructeur1(…) : constructeur2 (…)***

- Le premier constructeur va déléguer une partie du travail au second constructeur.

La délégation de constructeurs : exemple

- Ici, la classe Shape a deux constructeurs.
- Le second constructeur délègue une partie de la tâche au premier constructeur.
- **Attention ! Vous ne pouvez pas initialiser directement un attribut membre dans un constructeur qui délègue à un autre constructeur !**

```
class Shape {
    public:
     int _x;
     int _y;

     Shape(int x) {
         _x = x;
     }
     Shape(int x, int y) : Shape(x) {
         _y = y;
     }
};
```

Les énumérations “type safe”

- Les **enums de type safe**, aussi appelé **classe d’enums**, sont des énumérations typés et ayant une portée spécifique.

- Contrairement aux enums classiques, les classes d’enums sont **accessibles uniquement à partir du nom de l'énumération**. Cela permet d’éviter des conflits de noms. De plus il n’y a **pas de conversion implicite de leurs valeurs en int**.

```
enum class Color { Red, Green, Bleu, Yellow };

Color red = Color::Red;
```

Le mot-clé « auto »

- A partir de C++ 11, un des nouveaux mot-clés à faire son apparition est « ***auto*** ».

- Ce mot-clé permet d'indiquer que le type d'une variable sera déduit automatiquement à sa déclaration par le compilateur.

- Exemple :

```
int a = 12;          // a est explicitement de type int
auto b = 10;         // b est implicitement de type int

std::cout << typeid(a).name() << std::endl;
std::cout << typeid(a).name() << std::endl;
```

La boucle sur un intervalle

- La boucle basée sur un intervalle est une autre manière de parcourir un ensemble d'éléments.

- Elle s'écrit selon la syntaxe suivante :
  - ***for (type elementName: container) { … }***
- Exemple :

```
std::vector<std::string> cities { "Paris", "Londres", "Berlin", "Rome" };
for(std::string str: cities) {
    std::cout << str << std::endl;
}
```

Les références rvalue

- Une **rvalue** désigne une valeur non-nommée temporaire qui n'existe que pendant l'évaluation d'une expression. Elles sont principalement utilisées par les **constructeurs de déplacement**.

- Exemple :
  - int x = 1 + 2
- « 1 + 2 » est une valeur temporaire existant uniquement lors de l'opération d'initialisation. C'est donc une rvalue.

Les constructeurs de déplacement

- Le **constructeur de déplacement** permet de transférer des données d'un objet vers un autre.

- Cela implique de passer une **référence sur la rvalue** du second objet. On attribuera ensuite les valeurs à l'objet que l'on souhaite instancier et on réinitialisera celles du second objet.

- Enfin pour appeler le constructeur de déplacement, il faut utiliser la fonction ***std::move*** qui retourne une **rvalue**.

Impact sur la forme normale des classes

- Lorsqu'on utilise des **pointeurs**, un **constructeur de déplacement** peut simplement "déplacer" les données en s'appropriant les pointeurs faisant référence aux données.

- Le nouvel objet pointe désormais sur les données d'origine, et l'objet source est modifié pour ne plus pointer sur ces données (qui elles restent intactes).

- La mise à jour des pointeurs sur les données **est plus rapide que la copie** des données. Il n'y a pas besoin de copier les données puis de détruire les originales. Cela permet de **réduire l'utilisation de la mémoire** par rapport à un constructeur par copie.

Exemple d'utilisation d'un constructeur de déplacement

- Le gain de performance d'un constructeur de déplacement par rapport à un constructeur par copie est beaucoup plus important !

- Exemple :

```
People p1("Aline");
People p2(std::move(p1));

std::cout << p2.getName() << std::endl;
```

```
class People {
    private:
     std::string name;

    public:
     People(std::string n) {
         name = n;
     }
     People(People&& other) {
         std::cout << "Move constructor" << std::endl;
         this->name = other.name;
         other.name = "";
     }
     std::string getName() {
         return name;
     }
};
```

Les expressions lambda

- Les **lambdas** sont des **fonctions anonymes** définies et appelées à l'emplacement où elles sont déclarées.

- Elles permettent **d'encapsuler du code** et sont très utilisées dans les algorithmes et en programmation asynchrone.

- On reconnaît les lambdas à la syntaxe suivante :
    - ***[](param1, param2, …) { /* instructions */ }***

Les expressions lambda : exemple

```cpp
int main() {
    std::vector<std::string> mots = {"chien", "chat", "poulet", "ornithorynque", "lion"};

    std::sort(mots.begin(), mots.end(), [](const std::string& a, const std::string& b) {
        return a.length() < b.length();
    });

    for (const std::string& mot : mots) {
        std::cout << mot << " ";
    }
    return 0;
}
```

Points essentiels à retenir

- C++ 11 a permis de nombreuses améliorations au C++, comme par exemple :
  - l'apparition du pointeur **nullptr**
  - l'ajout des directives **delete** et **default**
  - la possibilité d'utiliser la **délégation de constructeur** et des constructeurs de déplacement
  - la présence **d'enum de type safe**
  - l'auto assignation du type d'une variable avec le mot-clé **auto**
  - la boucle **for sur un intervalle**
  - l'utilisation des **lambdas**.

IV – La gestion des opérateurs

Les opérateurs binaires et unaires

Symbole	Signification
&&	ET logique
||	OU logique
!=	Différent/NON/contraire à
!	Négation logique
&	ET bit à bit
|	OU inclusif bit à bit
^	OU exclusif bit à bit
~	Inversion binaire

L’opérateur d’indirection

- **L’opérateur d’indirection**, aussi appelé **opérateur de déréférencement**, « * » est utilisé pour déclarer des **pointeurs**.

- Il sert aussi à déréférencer un pointeur, c’est à dire à accéder à la valeur stockée à l’adresse mémoire pointée par le pointeur.

```
int value = 42;
int *ptr = &value;

std::cout << *ptr << std::endl;
```

L’opérateur de référencement

- Une **référence** est une variable, reconnu au symbole « **&** », qui fait « référence » à une autre variable existante. On peut donc modifier la valeur d’une variable à travers celle-ci ou à travers sa référence.

- Bien que similaire aux **pointeurs**, une référence ne peut pas être **NULL**, doit être **initialisée** à sa création et ne peut pas être modifiée pour faire référence à une autre variable.

```
int a = 42;
int& r = a;

std::cout << "Valeur de a : " << a << std::endl;
std::cout << "Valeur de r : " << r << std::endl;
a = 21;
std::cout << "Valeur de a : " << a << std::endl;
std::cout << "Valeur de r : " << r << std::endl;
```

Incrémentation et décrémentation préfixées et post-fixées

- L'**incrémentation préfixée** permet d'incrémenter la valeur d'une variable **avant** de l'utiliser dans une expression.
- L'**incrémentation post-fixée** permet d'incrémenter la valeur d'une variable **après** l'avoir utilisée dans une expression.

Symbole	Signification
x++	Incrémentation post-fixée de la valeur x
++x	Incrémentation préfixée de la valeur x
x--	Décrémentation post-fixée de la valeur x
--x	Décrémentation préfixée de la valeur x

Les opérateurs de comparaison

Symbole	Signification
==	Est égal à
!=	Est différent de
<	Est strictement inférieur à
<=	Est inférieur ou égal à
>	Est strictement supérieur à
>=	Est supérieur ou égal à

Les opérateurs d'affectation

Symbole	Signification
=	Enregistre la valeur
+=, -=, *=, /=, %=	Effectue l'opération adéquate et enregistre la valeur
<<=	Effectue un décalage de bit sur la gauche et enregistre la valeur
>>=	Effectue un décalage de bit sur la droite et enregistre la valeur
&=	Effectue un ET binaire et enregistre la valeur
^=	Effectue un OU exclusif et enregistre la valeur
|=	Effectue un OU inclusif et enregistre la valeur

La surcharge de l'opérateur []

- L'opérateur d'indice « **[]** » permet d'accéder aux éléments d'un tableau et il peut aussi être surchargé.
- Ici la classe SafeArray surcharge cette opérateur pour éviter les erreurs « *out of bounds* ».

```cpp
int main() {
    SafeArray arr;

    std::cout << "Value of arr[2] : " << arr[2] << std::endl;
    std::cout << "Value of arr[12] : " << arr[12]<< std::endl;
    return 0;
}
```

```cpp
const int SIZE = 10;

class SafeArray {
    int arr[SIZE];

  public:
    SafeArray() {
        for(int i = 0; i < SIZE; i++) {
            arr[i] = i;
        }
    }

    int &operator[](int i) {
        if (i > SIZE || i < 0) {
            std::cout << "Index out of bounds" << std::endl;
            return arr[0];
        }
        return arr[i];
    }
};
```

Surcharge des opérateurs ‘<<’ et ‘>>’

- Les différents flux d’entrées/sorties peuvent aussi être redirigés à travers les opérateurs « **<<** » et « **>>** ».
- Comme ci-dessous afin de modifier l’affichage d’un objet Date :

```
int main() {
    Date dt(24, 4, 23);
    std::cout << dt << std::endl;
}
```

```
class Date {
    int month, day, year;

  public:
    Date(int d, int m, int y) {
        day = d;
        month = m;
        year = y;
    }
    friend std::ostream& operator<<(std::ostream& os, const Date& dt);
};

std::ostream& operator<<(std::ostream& os, const Date& dt) {
    os << dt.day << '/' << dt.month << '/' << dt.year;
    return os;
}
```

Les foncteurs et la surcharge de l'opérateur ()

- Les **foncteurs** sont des classes qui se comportent comme des **fonctions**. Très souvent, ces classes **surchargent l'opérateur ()** afin d'être utilisées comme des fonctions.

- Ils permettent de simplifier certaines tâches et sont très utilisés dans la STL car ils peuvent être passé en tant qu'argument à d'autres fonctions tout en étant très efficients.

```
class Increment {
private:
    int num;
public:
    Increment(int n) : num(n) {  }

    int operator()(int value) const {
        return num + value;
    }
};

int main() {
    Increment addTen(10);

    int result = addTen(2);
    std::cout << result << std::endl;
    return 0;
}
```

Points essentiels à retenir

- Il existe de **nombreux opérateurs** en C++, que ce soit pour faire des opérations binaires, des décalages de bits ou bien simplifier des calculs.
- Ces opérateurs peuvent être **surchargés** afin de modifier leur comportement !
- L'opérateur d'indirection ***** est utilisé avec les **pointeurs**.
- L'opérateur de référencement **&** est utilisé pour les **références**.
- L'ordre de l'incrémentations : ++i et i++ a une importance.
- Il est possible d'utiliser des objets comme si c'était des fonctions, ce sont les **foncteurs**.

V – Conversion et RTTI

Les opérateurs de conversion

- En C++, les conversions peuvent se faire de manière **implicite** ou **explicite**, comme ci-dessous :

```
int i = 10;
double result = (double) i / 3;
```

- De plus, quatre principaux types de cast existent en C++ afin de modifier le type d'une donnée :
  - le **static_cast**
  - le **const_cast**
  - le **dynamic_cast**
  - et **le reinterpret_cast**.

Les opérateurs de casting - 1

- En C++, les conversions peuvent se faire de manière implicite ou explicite.

```
int i = 10;
double result = (double) i / 3;
```

- Plusieurs types de cast existent en C++ afin de modifier le type d'une donnée.

- ***static_cast<type> (expr)*** : L'opérateur « ***static_cast*** » effectue une conversion classique entre deux types de données. C'est peut-être l'opérateur de cast le plus utilisé, privilégiez-le !

Les opérateurs de casting - 2

- ***reinterpret_cast<type> (expr)*** : L'opérateur « ***reinterpret_cast*** » transforme un pointeur en n'importe quel autre type de pointeur. Il permet aussi de convertir un pointeur en un type d'entier et inversement. Très puissant !

- ***const_cast<type> (expr)*** : L'opérateur « ***const_cast*** » est utilisé pour remplacer explicitement « ***const*** » dans un cast. Le type cible doit être identique au type source.

```
float f = 3.5;
int value = static_cast<int>(f);
std::cout << "Value : " << value << std::endl;
```

Les conversions dynamiques

- L'opérateur ***dynamic_cast<type> (expr)*** permet d'effectuer une conversion dynamique à l'exécution et vérifie la validité du cast.

- Si le cast ne peut pas être effectué, alors il échoue et l'expression est évaluée à **nulle** dans le cas d'un pointeur, ou bien il lève une exception **std::bad_cast** dans le cas d'une référence.

- Les conversions dynamiques sont très utilisées pour travailler avec **l'héritage** des classes.

- Certains compilateurs ne supportent pas les conversions dynamiques.

Les conversions dynamiques : exemple

- Exemple de conversion dynamique d'une classe dérivée vers une classe de base.

- Il s'agit ici d'un « **upcast** » car le pointeur est déplacé vers le haut de la hiérarchie de classes.

```cpp
class Base {
public:
    Base() {};
    virtual ~Base() {};
};

class Derived : public Base {
public:
    Derived() {};
    void display() { std::cout << "Fonction dérivée" << std::endl; }
};

int main() {
    Derived derived;
    Base *ptr = dynamic_cast<Base*>(&derived);

    if (ptr) {
        std::cout << "Conversion réussie" << std::endl;
    }
    return 0;
}
```

Runtime Type Information

- Le **runtime type information** (**RTTI**) est un mécanisme permettant de déterminer le type d'un objet pendant l'exécution du programme. Ce mécanisme s'applique aux pointeurs et aux références.

- Il existe trois éléments principaux pour le RTTI :
  - L'opérateur **dynamic_cast** : utilisé pour les conversions dynamiques.
  - L'opérateur **typeid** : qui permet d'identifier le type d'un élément.
  - La classe **type_info** : qui est utilisé pour contenir les informations de type renvoyées par typeid.

L’opérateur typeid et les exceptions liées

- L’opérateur **typeid** permet de connaître le type d’un objet ou d’une variable. Il retourne un objet de type « ***type_info*** ».

- Il dispose d’une méthode ***name()*** permettant d’obtenir le nom du type de variable.

- Enfin si une **expression nulle ou incomplète** est passée à typeid, une **exception** de type ***bad_typeid*** peut être levée. Si le pointeur ne pointe pas vers un objet valide, une exception ***__non_rtti_object*** est levée.

L’opérateur typeid et les exceptions liées : exemple

```cpp
class Base {};
class Derived : public Base {};

int main() {
    int num = 404;
    double real = 3.21;
    std::string str = "Hello";
    Base baseObject;
    Derived derivedObject;

    std::cout << "Type de l'entier : " << typeid(num).name() << std::endl;
    std::cout << "Type du reel : " << typeid(real).name() << std::endl;
    std::cout << "Type de la chaine : " << typeid(str).name() << std::endl;
    std::cout << "Type de l'objet de base : " << typeid(baseObject).name() << std::endl;
    std::cout << "Type de l'objet derive : " << typeid(derivedObject).name() << std::endl;

    if (typeid(derivedObject) == typeid(Derived)) {
        std::cout << "derivedObject est de type Derived." << std::endl;
    }
    return 0;
}
```

La classe type_info

- La classe **type_info** est ainsi utilisée par **typeid** et accessible à partir de l'entête ***<typeinfo>***. Elle contient des informations sur les types de données utilisés dans le programme.

- Cette classe ne dispose que d'un constructeur de copie privée et donc on ne peut instancier des objets type_info qu'à travers l'utilisation de typeid.

```
int number = 42;

const std::type_info& infoNum = typeid(number);
std::cout << "Type : " << infoNum.name() << std::endl;
```

Downcasting avec dynamic_cast

- Le **downcasting** consiste à convertir un pointeur ou une référence d'une classe de base vers une classe dérivée.

- Le downcasting permet ainsi d'accéder aux méthodes spécifiques des classes dérivées.

```cpp
class Base {
public:
    Base() {};
    virtual ~Base() {};
};

class Derived : public Base {
public:
    Derived() {};
    void display() { std::cout << "Fonction dérivée" << std::endl; }
};

int main() {
    Base* ptrBase = new Derived();
    Derived* ptrDerived = dynamic_cast<Derived*>(ptrBase);

    if (ptrDerived) {
        ptrDerived->display();
    }
    return 0;
}
```

Points essentiels à retenir

- Il existe différents types de conversions en C++ : les conversions **implicites** et **explicites**.
- Le mot-clé ***explicit*** peut être utilisé pour interdire à un constructeur de faire des conversions implicites.
- On trouve quatre grands types d'opérateurs de conversion en C++ : ***static_cast***, ***dynamic_cast***, ***reinterpret_cast***, ***const_cast***.
- L'opérateur ***typeid*** permet d'obtenir un objet ***type_info*** qui contient des informations sur le type de données utilisé.
- **Dynamic_cast** permet de faire des ***downcast*** et des ***upcast***.

VI – La généricité

Introduction aux patrons de classes

- Les **patrons**, aussi appelés **templates** ou **modèles**, sont des **classes**, **fonctions**, **opérateurs** qui peuvent traiter des données de **type générique** : ils permettent d'écrire du code indépendant de tout type de données spécifique.

- Ainsi on peut utiliser la même classe aussi bien avec des données de type « *int* » que « *string* » par exemple. Cela permet d'avoir un code beaucoup plus facilement **maintenable** et évite la duplication de code.

- Certaines classes de la **STL** utilisent extensivement les modèles, c'est le cas des conteneurs.

Généricité et préprocesseur

- La **généricité** est gérée par le **compilateur** C++. Il n'y a pas besoin de préprocesseur pour cela.

- Le **préprocesseur** traite les fichiers sources **avant le compilateur**. Il n'a donc **pas connaissance des types** utilisés par les modèles génériques. C'est le compilateur qui va générer le **code spécifique à chaque type** utilisé lors de l'instanciation du modèle.

- C'est de cette manière que la généricité permet de rendre le code plus **flexible** et **réutilisable**.

Généralisation et composition générique

- La **généralisation générique**, aussi appelée **généricité**, est un concept où l'écrit du code pouvant fonctionner avec différents types de données. Cela permet d'avoir du code facilement **réutilisable**, **polyvalent** et **modulaire**.

- Ce mécanisme est appliqué à travers les **templates**, que ce soit sur des **fonctions**, des **classes** ou des **structures de données**. Les types de données traités par ces éléments sont spécifiés l'instanciation.

- La possibilité de créer des structures de données génériques en combinant différents composants génériques est ce que l'on appelle la **composition générique**.

Les fonctions génériques

- Les **fonction génériques** (***function templates***) permettent d'écrire des fonctions indépendamment du type de données utilisés en paramètres ou en valeur de retour.

- Ils respectent la syntaxe suivante :
  - ***template <class type> ret-type func-name(parameter) ;***

- Exemple :

```
template <typename T> T add(T a, T b) {
    return a + b;
}
```

Classe générique - 1

- Une **classe générique** (aussi appelée ***modèles de classe*** ou ***class templates***) permettent d'écrire des classe indépendamment du type de données utilisés.

- Elles offrent ainsi la possibilité de **surcharger** et **redéfinir** les méthodes de ces classes.

- Elles respectent la syntaxe suivante :
    - ***template <class type> class class-name { };***

Classe générique - 2

- Exemple d'une **classe générique** utilisant deux types de données.

- Les symboles « *T* » et « *M* » représentent ici des **types de données génériques** pouvant être remplacés par n'importe quel autre type de données.

```
template<class T, class M> class MyClass {

    T param1;
    M param2;

  public:
    MyClass(T p1, M p2) {
        param1 = p1;
        param2 = p2;
    }

    T getParam() {
        return param1;
    }
    void setParam(T p) {
        param1 = p;
    }
    M getParam() {
        return param2;
    }
    void setParam(M p) {
        param2 = p;
    }
};
```

Généricité et surcharge des opérateurs

- Les opérateurs peuvent aussi être surchargés en utilisant la généricité.

```
template<class T> class Container {
    T val;

  public:
    void setVal(T a) {
        val = a;
    }

    Container<T> operator+(Container<T>& other) {
        Container<T> result;
        result.val = this->val + other.val;
        return result;
    }
};
```

```
int main() {
    Container<int> a;
    Container<int> b;
    Container<int> c;
    a.setVal(42);
    b.setVal(3);
    c = a + b;
    return 0;
}
```

Généricité et les mécanismes de dérivation

- La dérivation reste possible avec l'utilisation de modèles.

- La subtilité consiste à appeler les méthodes de la classe parente avec la syntaxe :
  - ***ParentClass<T>::childrenMethod();***

```
template<class T> class Base {
    T val;

  public:
    void setVal(T a) {
        val = a;
    }
};

template<class T> class Derived : public Base<T> {
  public:
    void setVal (T b) {
        Base<T>::setVal(b);
    }
};

int main() {
    Derived<int> a;
    a.setVal(42);
    return 0;
}
```

Spécialisation totale

- La **spécialisation totale** permet de personnaliser le comportement d'un template pour un type de données spécifique.

- Ce type de donnée peut ainsi avoir une implémentation complètement différente du modèle générique.

- En fonction du type spécifié, le compilateur choisira la méthode la plus appropriée.

```
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}

template <>
char max<char>(char a, char b) {
    return std::toupper(a) > std::toupper(b) ? a : b;
}

int main() {
    std::cout << max(12, 4) << std::endl;
    std::cout << max('z', 'c') << std::endl;
    return 0;
}
```

Spécialisation partielle

- La **spécialisation partielle** permet de personnaliser seulement une partie d'un modèle pour un type de données spécifique, le reste restant générique.

- La spécialisation totale et partielle sont deux des notions avancées de la généricité.

```cpp
template <typename T> class Container {
public:
    void display() const {
        std::cout << "Modèle principal" << std::endl;
    }
};

template <typename T> class Container<T*> {
public:
    void display() const {
        std::cout << "Spécialisation partielle " << std::endl;
    }
};

int main() {
    Container<int> containerInt;
    Container<int*> containerPtr;

    containerInt.display();
    containerPtr.display();
    return 0;
}
```

Introduction à la méta-programmation - 1

- La **méta-programmation** consiste à utiliser les templates pour **exécuter du code pendant la compilation**. Cela permet d'effectuer des tâches complexes lors de la compilation, plutôt qu'à l'exécution afin d'améliorer les **performances** du programme.

```
int factorial(unsigned int n) {
    if (n == 0)
        return 1;
    return n * factorial(n - 1);
}
```

Fonction factorielle récursive

```
template <unsigned int N>
struct Factorial {
    enum { value = N * Factorial<N - 1>::value };
};

template <>
struct Factorial<0> {
    enum { value = 1 };
};
```

Utilisation de la méta-programmation avec des templates

Introduction à la méta-programmation - 2

- Ainsi on peut utiliser une **méthode** ou un **objet générique** afin d'effectuer des opérations lors de la compilation et d'obtenir le résultat à l'exécution du programme.

```
int main() {
    int result = Factorial<4>::value;
    std::cout << result << std::endl;
}
```

- Parmi les inconvénients de la méta-programmation, on notera la **dépendance au compilateur**. Changer ce dernier peut créer des **problèmes de portabilité**. De plus, la syntaxe de la méta-programmation n'est pas des plus **lisibles**, en particulier pour les développeurs juniors.

La généricité, principe fédérateur des librairies STL et Boost

- Une des applications les plus communes de la généricité se trouvent au sein des **librairies STL** et **Boost**.

- Comme cela sera vu plus tard, ces librairies disposent de nombreuses méthodes, algorithmes et objets génériques.

- Que ce soit les **conteneurs** et les **itérateurs** de la STL ou bien les **algorithmes** et **objets** de Boost.

Points essentiels à retenir

- La **généricité** permet de créer des **classes**, **structures** et **fonctions** pouvant utiliser n'importe quel type de données.
- Les classes et fonctions peuvent manipuler plusieurs types de données génériques.
- Les opérateurs peuvent être **surchargés** tout en étant génériques.
- L'utilisation de classes générique n'empêche pas la **dérivation**.
- La **spécialisation totale** ou **partielle** permet d'avoir un comportement spécifique selon un type de donnée particulier.
- La généricité est très utilisée dans les librairies STL et Boost.

VII – La STL

Aperçu sur la STL

- La **STL**, pour ***Standard Template Library***, est un ensemble de classes et méthodes permettant aux développeurs d'utiliser des classes de **stockage**, des **structures de données** génériques, des **algorithmes** populaires.
- Elle est composée de trois grandes parties :
    - Des **conteneurs**, tel que des listes chaînées, vecteurs, tableaux associatifs, etc.
    - Des **algorithmes** génériques permettant d'initialiser, trier, manipuler le contenu des conteneurs.
    - Des **itérateurs**, qui permettent de parcourir aisément des conteneurs génériques.

Objectifs et principes

- La STL a été développé avec l'idée de **séparer les algorithmes des structures de données**, les **itérateurs faisant le lien** entre les deux.
- Elle permet aussi aux développeurs de créer **leurs propres algorithmes et structures de données** en se basant sur des **modèles** prédéfinies.

Algorithmes	Structure de données
std::copy	std::vector
std::sort	std::map
…	…
Algorithme définie par l'utilisateur	Structures définie par l'utilisateur

Itérateurs

Les chaînes de caractères STL : la classe template basic_string

- La classe ***basic_string*** est une classe **générique** de manipulation de chaînes de caractères de n'importe quel type.

- Elle dispose d'itérateurs et de différentes méthodes pour manipuler son contenu :
  - ***size(), length(), clear(), empty(), append(), replace(), insert(), erase(), push_back(), swap(), pop_back()***, etc.
- Elle dispose de classes spécialisées permettant de gérer d'autres types de chaînes comme ***string***, ***wstring***, ***u16string***, ***u32string***.

Les conteneurs séquentiels et associatifs

- Les **conteneurs** sont des objets, implémentés en tant que modèle de classe, pouvant contenir une **collections d'éléments** ayant le même type. Ils offrent des méthodes permettant **d'accéder** et de **manipuler** ces éléments.

- On trouve trois grandes catégories de conteneurs :
  - Les **conteneurs de séquence** comme ***vector***, ***list***, ***deque***, etc.
  - Les **conteneurs associatifs** comme ***map***, ***set***, ***multimap*** ou encore ***multiset***.
  - Les **adaptateurs de conteneurs** comme ***queue***, ***priority_queue*** ou ***stack***.

Conteneurs de séquence vector et list

- Les **vecteurs** et les **listes** sont des conteneurs permettant de stocker des objets d'un type donné.

- Leur différence réside dans le fait qu'un **vecteur** permet de stocker ses éléments de manière contiguë en mémoire (à la manière d'un **tableau**) et est privilégié lorsque l'on souhaite **accéder facilement à ses éléments**.

- Là où la **liste** fonctionne à la manière **d'une liste doublement chaînée** et est plus efficace lorsque l'on effectue de **nombreux ajouts ou suppression d'éléments**.

Exemple d'utilisation d'un vecteur

- Les **vecteurs**, comme tous les conteneurs, disposent de nombreuses méthodes permettant d'effectuer de nombreuses actions aisément.

- *push_back(), at(), pop_back(), size(), empty(), clear(), begin(), end(), …*

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> num {1, 2, 3, 4, 5};

    for (const int& i : num) {
        std::cout << i << "  ";
    }
    num.push_back(16);
    num.push_back(7);
    num.at(5) = 6;
    std::cout << "Element à l'index 5 : " << num.at(5) << std::endl;
    num.pop_back();
    for (const int& i : num) {
        std::cout << i << "  ";
    }

    return 0;
}
```

Les conteneurs associatifs : map

```cpp
#include <iostream>
#include <map>

typedef std::map<const char*, int> Map;

int main() {
    std::map<const char*, int> shoppingList;

    shoppingList.insert(Map::value_type("Chocolat", 3));
    shoppingList.insert(Map::value_type("Banane", 8));
    shoppingList.insert(Map::value_type("Vin", 1));

    // find and show elements
    std::cout << shoppingList.begin()->first << " " << shoppingList.begin()->second << std::endl;
    std::cout << "shoppingList.at('Banane') : " << shoppingList.at("Banane") << std::endl;
    std::cout << shoppingList.find("Vin")->first << std::endl;
    return (0);
}
```

- Les ***map*** sont des conteneurs associatifs de taille variable fonctionnant sous la forme de **clé-valeur**, la clé étant unique.

Les adaptateurs de conteneurs : queue et stack

- D'autres types de conteneurs existent tel que les ***queue*** et les ***stack***.
- Une **queue** est une conteneur opérant dans un contexte « ***first-in first-out*** » où les éléments sont insérés à une extrémité et ressortent par l'autre.
- Syntaxe : ***template <class T, class Container = deque<T> > class queue;***

- Une **pile** est un conteneur opérant dans un contexte « ***last-in first-out*** » où les éléments sont insérés et extraits de la même extrémité.
- Syntaxe : ***template <class T, class Container = deque<T> > class stack;***

Exemple d'utilisation d'une stack

```cpp
#include <iostream>
#include <stack>

int main() {
   std::stack <int> myStack;
   std::stack <int>::size_type i;

   myStack.push(10);
   myStack.push(20);
   myStack.push(30);

   i = myStack.size();
   std::cout << "La taille de la stack est de : " << i << "." << std::endl;
   i = myStack.top();
   std::cout << "La valeur du dessus de la stack est " << i << "." << std::endl;
   myStack.pop();
   i = myStack.top();
   std::cout << "Après un pop, la valeur du dessus de la stack est " << i << "." << std::endl;

   return 0;
}
```

Les méthodes d'insertion, de suppression, d'itération et d'accès

- Tous ces conteneurs disposent d'un certain nombres de méthodes et d'itérateurs dont certains sont communs à plusieurs d'entre eux.

- **Vector** : *at(), front(), back(), assign(), push_back(), pop_back(), insert(), erase(), swap(), clear(), …*
- **Stack** : *empty(), size(), top(), push(), pop(), swap(), …*
- **List** : *empty(), size(), front(), back(), assign(), push_front(), pop_front(), push_back(), pop_back(), insert(), erase(), swap(), clear(), splice(), remove(), merge(), sort(), reverse(), …*
- **Set** : *empty(), size(), max_size(), insert(), erase(), swap(), clear(), find(), count(), equal_range(), …*

Les allocateurs et la gestion de la mémoire des conteneurs

- Les conteneurs de la STL peuvent prendre des **allocateurs** lorsqu'ils sont initialisés.

- Un **allocateur** permet d'**allouer**, de **libérer** et d'**optimiser** la mémoire pour les éléments d'un conteneur. Ainsi un allocateur par défaut est initialisé lorsque l'on instancie un nouveau conteneur.

- Néanmoins il est aussi possible de créer ses propres allocateurs qui doivent alors contenir :
    - un **constructeur par copie** et les méthodes ***operator==***, ***operator!=***, ***allocate()***, ***deallocate()***.

Le concept d'itérateur

- Les **itérateurs** sont des **objets** permettant **d'itérer** sur un conteneur et **d'accéder** à ses éléments grâce à différents opérateurs. Ils prennent souvent la forme de **pointeur**.

- Les conteneurs de la **STL** (*vector*, *list*, *map*, …) disposent tous d'itérateurs permettant d'accéder à leurs éléments de façon standard.

- Enfin, ils sont basés sur le modèle de classe de ***std::iterator*** accessible avec ***l'include <iterator>***.

Les types d'itérateurs

- On trouve cinq grands types d'itérateurs :
  - Les **itérateurs de sortie** - *output iterators*.
  - Les **itérateurs d'entrée** - *input iterators*.
  - Les **itérateurs vers l'avant** - *forward iterator*.
  - Les **itérateurs bidirectionnels** - *bidirectional iterators*.
  - Les **itérateurs à accès aléatoire** - *random-access iterators*.

Les propriétés des itérateurs d'entrée et de sortie

Un **itérateur de sortie** :

- peut itérer vers l'avant sur un conteneur avec **l'opérateur** « ***++*** ».
- peut écrire un élément une seule fois avec **l'opérateur** « ***** ».

- Un **itérateur d'entrée** :
  - peut itérer vers l'avant sur un conteneur avec **l'opérateur** « ***++*** ».
  - peut lire un élément n'importe quel nombre de fois à l'aide de **l'opérateur** « ***** » .
  - peut être comparer à un autre itérateur d'entrée avec les **opérateurs** « ***!=*** » et « ***==*** ».

Les propriétés des itérateurs vers l'avant

- Un **itérateur vers l'avant** :
  - peut itérer vers l'avant sur un conteneur avec **l'opérateur** « ***++*** ».
  - peut lire ou écrire des éléments non « *const* » avec l'opérateur « ***** » .
  - permet d'accéder aux membres d'un élément avec de **l'opérateur** « ***->*** ».
  - peut être comparé à un autre itérateur avant avec les **opérateurs** « ***!=*** » et « ***==*** ».
  - peut être copié. Chaque copie pouvant être déréférencée et incrémentée de manière indépendante.

Les propriétés des itérateurs bidirectionnels

- Un **itérateur bidirectionnel** :
  - peut être utilisé à la place d'un **itérateur vers l'avant**, il dispose de toutes ses propriétés.
  - peut être décrémenté avec **l'opérateur** « **--** ».
  - permet d'accéder aux membres d'un élément avec de **l'opérateur** « **->** ».
  - peut être comparé à un autre itérateur bidirectionnel avec les **opérateurs** « ***!=*** » et « ***==*** ».

Les propriétés des itérateurs à accès aléatoires

- Un **itérateur à accès aléatoire** :
  - peut être utilisé à la place d'un **itérateur bidirectionnel**.
  - permet d'utiliser l'opérateur d'indice « ***[]*** » pour accéder aux éléments.
  - supporte les opérations arithmétiques avec les opérateurs « ***+*** », « ***-*** », « ***+=*** » et « ***-=*** ».
  - peut être comparé à d'autres des itérateurs bidirectionnels avec les **opérateurs** « ***==*** », « ***!=*** », « ***<*** », « ***>*** », « ***<=*** » et « ***>=*** ».

Les méthodes des itérateurs

- Quelques méthodes utiles apportées par les itérateurs :

Fonctions	Description
begin()	Retourne un itérateur pointant sur le premier élément d'un conteneur.
end()	Retourne un itérateur pointant **après** le dernier élément d'un conteneur.
prev(iterator, n)	Retourne un itérateur pointant sur l'élément que *iterator* pointerait s'il était avancé de *-n* positions.
next(iterator, n)	Retourne un itérateur pointant sur l'élément que *iterator* pointerait s'il était avancé de *n* positions.
advance(iterator, n)	Avance l'itérateur passé en paramètre de *n* positions.
distance(first, last)	Donne le nombre d'élément entre les itérateurs *first* et *last*.

Exemple d'utilisation d'un itérateur

```cpp
#include <iterator>
#include <vector>
#include <iostream>

int main()
{
    std::vector<int> vec;

    for (int i = 0; i < 10; i++) {
        vec.push_back(i);
    }

    for (std::vector <int>::iterator iter = vec.begin(); iter != vec.end(); iter++) {
        std::cout << "Valeur : " << *iter << std::endl;
    }
}
```

Les différents groupes d'algorithmes STL

Algorithmes non-mutants	Algorithmes mutants	Algorithmes de tri et de fusion	Algorithmes numériques
Ne modifient pas les données qu'ils manipulent.	Modifient les données manipulées.	Trient et fusionnent les éléments qu'ils manipulent.	Effectuent des opérations mathématiques sur les valeurs manipulées.
std::find, std::count, std::all_of, std::any_of, std::none_of, std::for_each, std::_search, …	*std::sort, std::copy, std::remove, std::transform, std::fill, …*	*std::sort, std::stable_sort, std::partial_sort, std::merge, …*	*std::accumulate, std::partial_sum, std::iota, std::inner_product, …*

Paramétrer les algorithmes génériques par des objets fonctions

- Certains des algorithmes génériques de la STL peuvent prendre en paramètre un **objet** ou une **fonction**.

- C'est le cas de l'algorithme *std::sort* qui effectue un tri par ordre croissant mais peut aussi effectuer ce tri selon une fonction ou un objet passé en paramètre.

```
bool myFunction (int i, int j) { return (i < j); }

class MyClass {
public:
    bool operator()(int i, int j) { return (i < j); }
};

int main () {
    int tabInt[] = { 33, 71, 100, 12, 45, 26, 90, 57, 32 };
    std::vector<int> v(tabInt, tabInt+9);
    MyClass myClass;

    std::sort(v.begin(), v.begin());
    std::cout << "Defaut :";
    display(v);

    std::sort(v.begin(), v.end(), myFunction);
    std::cout << "Fonction :";
    display(v);

    std::sort(v.begin(), v.end(), myClass);
    std::cout << "Objet :";
    display(v);
    return 0;
}
```

La STL et les traitements sur les flux

- La STL offre plusieurs classes permettant de manipuler les **entrées et sorties** (**Input/Output**). Ces **I/O** sont gérées sous formes de **flux d'octets** appelés **streams**.

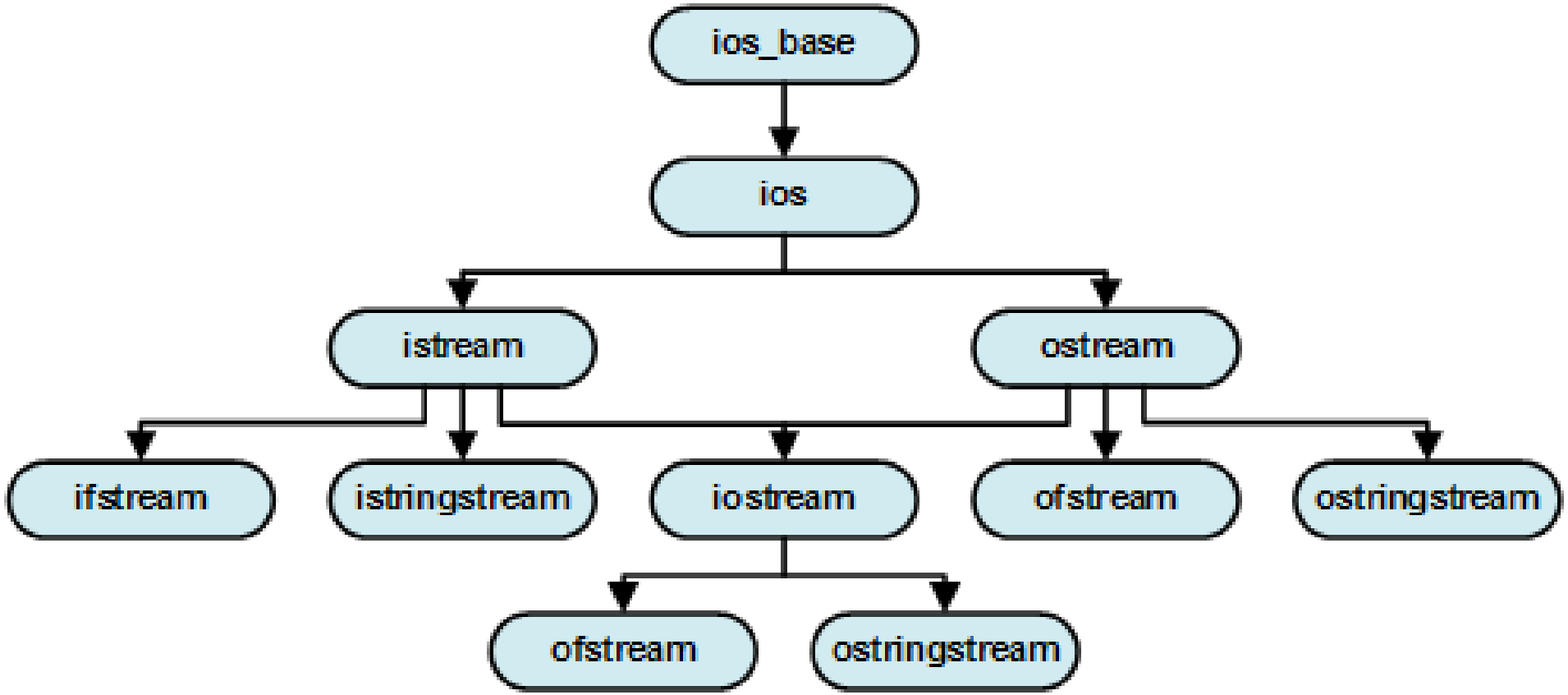
- Si ces flux de données viennent d'un périphérique comme le clavier ou une connexion réseau, ce sont des **entrées**.

- S'ils viennent de la mémoire vers un périphérique comme un écran, un disque ou une connexion réseau, ce sont des **sorties**.

Les principales classes d’I/O

- La plupart des classes d’I/O viennent de trois principaux fichiers d’entêtes :
  - ***<iostream>*** : définit notamment les objets ***cin***, ***cout***, et ***cerr***, qui correspondent au flux d'entrée standard, au flux de sortie standard et au flux de sortie d'erreur.
  - ***<iomanip>*** : définit des fonctions permettant de manipuler des entrées et sorties afin d’en formater les données.
  - ***<fstream>*** : définit des fonctions et objets pour manipuler les fichiers.

La hiérarchie des classes d'entrée/sortie



Description de quelques classes d'entrées/sorties

- De la même manière que la lecture/écriture sur *stdin* et *stdout*, on va utiliser des **streams** pour manipuler les fichiers en C++.

- ***Ofstream*** : est une classe qui permet de créer et d'écrire dans des fichiers.
- ***Ifstream*** : est une classe qui permet de lire les informations d'un fichier.
- ***Fstream*** : cette classe représente le stream du fichier et peut aussi bien créer, lire ou écrire dans des fichiers.

Exemple de manipulation de fichier

- Pour ouvrir un fichier on utilisera la fonction ***open()*** d'un stream, qui prend en paramètre le nom du fichier.

- A la fin, on utilise la fonction ***close()*** du stream pour fermer le fichier.

- Exemple d'écriture dans un fichier en utilisant **ofstream** :

```
#include <fstream>

int main() {
    std::string str = "Some data to write ...";

    std::ofstream outfile;
    outfile.open("myFile.txt");
    outfile << str << std::endl;
    outfile.close();
    return 0;
}
```

Les principes du RAII

- Le **RAII** (***Resource Acquisition Is Initialization***) est une technique qui consiste **à lier une ressource à la durée de vie d'un objet** de façon à s'assurer que cette ressource soit bien libérée lors de la destruction de l'objet.

- Ainsi un espace mémoire peut être alloué lors de la construction d'un objet et libéré à sa destruction. Cela mena à l'apparition des **pointeurs automatiques** et la classe ***auto_ptr***.

- Le but d'***auto_ptr*** est de détruire l'élément qu'il pointe et de libérer sa mémoire lors de sa destruction. Mais son incapacité à gérer les conteneurs de la STL et des problèmes de sécurité l'ont rendu **déprécié** et fait remplacer par ***unique_ptr***.

Les exceptions standards de la STL

- La STL met à disposition la classe ***std::exception*** incluse dans l'entête ***<exception>***. Cette classe est héritée par toutes les exceptions spécialisés de la STL et permet ainsi de toutes les gérer directement ou indirectement.

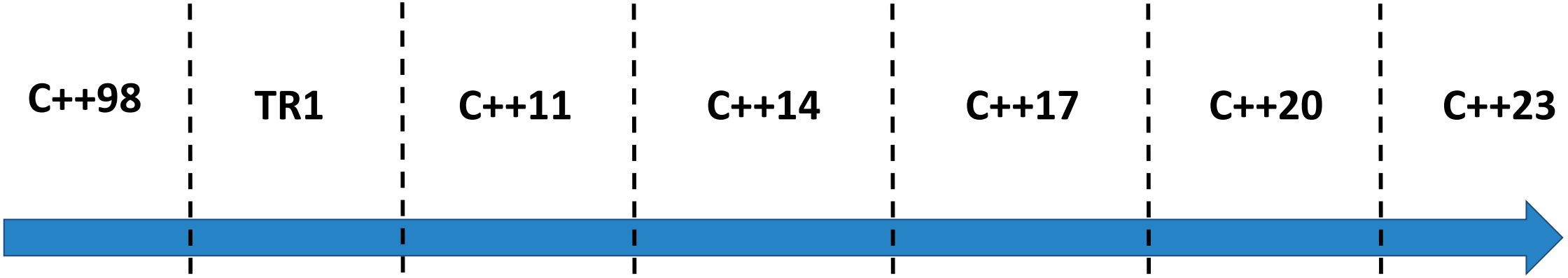
- Parmi ces exceptions on retrouve : *bad_alloc, bad_cast, bad_exception, bad_function_call, bad_typeid, bad_weak_ptr, ios_base::failure, logic_error, runtime_error …*

Points essentiels à retenir

- La **STL** apporte de nombreux éléments au C++ : des **conteneurs**, des **algorithmes**, des **itérateurs**, des **fonctions**, des **objets**, etc.
- Parmi les conteneurs on retrouve des **conteneurs associatifs** (map), **séquentiels** (vecteur, list), et **adaptatifs** (queue, stack).
- Les **itérateurs** permettent de parcourir et manipuler ces conteneurs facilement et efficacement.
- Différents **algorithmes** permettent de manipuler, modifier, trier et effectuer des opérateurs sur les éléments des conteneurs de la STL.
- Elle disposent d'objets permettant de manipuler des flux I/O : **Ofstream**, **Ifstream**, **Fstream**.
- Enfin elle dispose de ses propres classes exceptions avec **std::exception et ses dérivées**.

VIII – Les nouveautés de la librairie standard

Evolution historique



Premier standard ISO :
- opérateurs de conversion
- RTTI
- la STL propose des conteneurs, algorithmes et itérateurs

Technical Report 1 :
- regex
- hash tables
- tuple
- plusieurs fonctions mathématique

Seconde standard ISO :
- algorithmes
- smart pointers
- multithreading

Révision mineure du standard :
- amélioration des smart pointers et du multithreading
- ajout des lambdas génériques

Révision majeure du standard ISO après C++11 :
- inclusion d'éléments de bibliothèques d'extension (std::any, std::optional, std::variant, std::apply, etc)

Révision majeure du standard ISO après C++17 :
- coroutine
- module
- abréviation des fonctions template
- contraintes et concepts
- macros de test

Révision mineure de C++ qui prépare C++ 26 :
- évolution de la librairie standard std
- beaucoup de petites améliorations

Les nouveaux conteneurs - 1

- C++11 marque aussi l'apparition de nouveaux conteneurs :
  - **array** : un tableau de taille fixe qui stocke un nombre fixe d'éléments d'un type spécifique à la suite en mémoire. La taille du tableau est spécifiée lors de la déclaration et ne peut pas être modifiée. Entête : ***<array>***.

```
std::array<int, 5> myArray = {1, 2, 3, 4, 5};
```

  - **forward_list** : une liste simplement chaînée. Entête : ***<forward_list>***.

```
std::forward_list<int> myForwardList = {1, 2, 3, 4, 5};
```

Les nouveaux conteneurs - 2

- **unordered_set** : un conteneur associatif qui stocke un ensemble d'éléments uniques, de manière désordonnée. Entête : ***<unordered_set>***.

```
std::unordered_set<int> mySet = {1, 2, 3, 4, 5};
```

- **unordered_map** : un conteneur associatif qui stocke des paires clé-valeur. Il permet un accès rapide à ses éléments. Entête : ***<unordered_map>***.

```
std::unordered_map<std::string, int> myMap = {{"apple", 5}, {"banana", 3}, {"cherry", 8}};
```

La classe tuple

- Un **tuple** est une collection de taille fixe de plusieurs éléments de types différents. Elle nécessite l'entête ***<tuple>***.

- Cette classe dispose de plusieurs méthodes parmi lesquelles :
  - ***make_tuple()*** : assigne les valeurs passées en paramètre à un tuple.
  - ***tuple_cat()*** : crée un tuple en concaténant deux tuples.
  - ***get()*** : permet d'accéder à l'élément spécifié par le tuple.
  - ***swap()*** : échange le contenu de deux tuples.

La classe tuple : exemple

```cpp
#include <iostream>
#include <tuple>

int main() {
    std::tuple<int, std::string, float> myTuple;
    // std::tuple<int, std::string, float> myTuple(42, "Hello", 3.14);

    myTuple = std::make_tuple(42, "Hello", 3.14);
    int intValue = std::get<0>(myTuple);
    std::string stringValue = std::get<1>(myTuple);
    float floatValue = std::get<2>(myTuple);
    std::cout << intValue << " " << stringValue << " " << floatValue << std::endl;

    return 0;
}
```

- Ici on génère le tuple avec *make_tuple()* mais on aurait pu aussi bien le faire à l'initialisation.

Les pointeurs intelligents - 1

- Pour faciliter la gestion de la mémoire, on peut désormais utiliser des **pointeurs intelligents** (**smart pointer**) qui vont permettre de **gérer automatiquement la mémoire**. Ils permettent la libération automatique de celle-ci afin **d'éviter les fuites mémoires** !

- L'entête ***<memory>*** permet d'utiliser 3 types de pointeurs intelligents :

- ***unique_ptr*** : autorise uniquement **un seul propriétaire** au pointeur. Peut être déplacé vers un nouveau propriétaire, mais pas copié ou partagé. Sert à gérer des ressources uniques.

Les pointeurs intelligents - 2

- ***shared_ptr*** : peut avoir plusieurs propriétaires et permet de connaître le nombre de propriétaires qu'il a à travers ***std::shared_ptr***. L'objet pointé est détruit lorsque le dernier *std::shared_ptr* qui le possède est détruit. Sert à gérer des ressources partagées. Il contient le pointeur et la référence du nombre de propriétaires.

- ***weak_ptr*** : à utiliser avec avec *shared_ptr*. Permet l'accès à un objet appartenant à une ou plusieurs instances de *shared_ptr*, mais sans participer au décompte des références. Permet d'éviter les références circulaires entre les *std::shared_ptr*. Attention, il ne garantit pas l'existence du pointeur ciblé !

Les pointeurs intelligents - 3

- La méthode ***reset()*** permet de libérer la mémoire. Néanmoins cette libération s'effectue aussi automatiquement à l'appel du destructeur.

- On peut changer le propriétaire d'un *unique_ptr* avec ***std::move()***.

```cpp
#include <iostream>
#include <memory>

int main() {
    std::unique_ptr<int> uniquePtr1 (new int(5));
    // A partir de C++ 14
    std::unique_ptr<int> uniquePtr2 = std::make_unique<int>(42);
    // Ne compile pas !
    // std::unique_ptr<int> ptr  = uniquePtr1;

    std::cout << *uniquePtr1 << std::endl;
    std::cout << *uniquePtr2 << std::endl;
    // Libère la mémoire de uniquePtr2
    uniquePtr2.reset();

    std::shared_ptr<int> shared = std::make_shared<int>(21);
    std::weak_ptr<int> weak = shared;
    // Conversion en shared_ptr
    std::shared_ptr<int> shared2 = weak.lock();

    std::cout << *shared << std::endl;
    std::cout << *shared2 << std::endl;
    *shared = 10;
    std::cout << *shared << std::endl;
    std::cout << *shared2 << std::endl;
    return 0;
}
```

Les nouveaux foncteurs et binders - 1

- Avec C++11 apparaissent de nouveaux **foncteurs** : les **binders**.

- Ils permettent d'associer **certains des arguments** d'une fonction à une valeur fixe.

- Cette valeur fixe est stockée à l'intérieur du foncteur et le reste des arguments peut être ensuite passé dynamiquement au moment de l'appel du foncteur.

Les nouveaux foncteurs et binders - 2

- Pour cela on utilise ***std::bind()*** qui prend en paramètre une fonction et ses paramètres.

- On peut spécifier aussi des paramètres dynamiques avec ***std::placeholders***.

- Cela retourne un **foncteur**.

```cpp
#include <iostream>
#include <functional>

int add(int a, int b) {
    return a + b;
}

int main() {
    auto addFunction = std::bind(add, 10, std::placeholders::_1);
    int result = addFunction(20);
    std::cout << "Resultat : " << result << std::endl;
    return 0;
}
```

Introduction à la gestion des threads

- En C++, le multithreading dispose d'un support intégré depuis **C++11** et peut désormais s'effectuer à partir de la classe ***std::thread*** utilisable à partir du header ***<thread>***.

- Les **threads** vont chacun représenter une partie du programme dans lequel du code va être **exécuté en parallèle**. En général, un programme dispose d'un thread principal et d'un ou plusieurs threads secondaires.

- Tous les threads partagent la **même zone mémoire** mais ils disposent chacun **d'une pile d'appels différente**.

Utiliser un thread

- Le constructeur d'un **thread** s'appelle avec ***std::thread*** et prend en paramètre **une fonction** (pouvant être une lambda) qui va s'exécuter dans le thread et **les paramètres** de cette fonction.

- L'appel à la méthode ***join()*** d'un thread permet de s'assurer que celui-ci se termine avant la fin du programme. Sinon cela pourrait occasionner une erreur.

- La méthode ***joinable()*** permet de savoir si le thread est actif ou s'il a fini d'exécuter sa tâche et peut ainsi être « joint ». Cette méthode renvoie un booléen.

Exemple d'utilisation de thread

```cpp
int main()
{
    std::vector<std::string> array = {"Hello", "World", "chaussette", "lorem", "lion"};

    std::thread t1([]() {
        for (int i = 0; i < 10; ++i)
        {
            std::cout << "i : " << (i * 2) << " ";
        }
    });
    std::thread t2(add_sequence, 10, 20);
    std::thread t3(display_vector, array);
    t1.join();
    t2.join();
    t3.join();

    return 0;
}
```

- Ici *add_sequence* et *display_vector* sont des méthodes créées par l'utilisateur.

La méthode swap

- La méthode **swap** de ***<thread>*** surcharge l'algorithme ***std::swap()*** et permet d'échanger l'état de deux threads. Exemple :

```
std::thread t1();
std::thread t2();

std::cout << "Avant thread 1 id : " << t1.get_id() << std::endl;
std::cout << "thread 2 id = " << t2.get_id() << std::endl;

t1.swap(t2);

std::cout << "Apres thread 1 id : " << t1.get_id() << std::endl;
std::cout << "Apres thread 2 id : " << t2.get_id() << std::endl;

t1.join();
t2.join();
```

Les expressions régulières - 1

- Il est possible d'utiliser des regex en C++ avec l'entête ***<regex>***. Cette entête donne accès à la constante ***std::regex_constants::syntax_option_type*** qui permet de choisir le type de syntaxe des regex à utiliser avec par exemple :
  - **ECMAScript** : par défaut, syntaxe similaire aux regex de JavaScript et NET
  - **basic** : expressions régulières POSIX basic ou BRE.
  - **extended** : expressions régulières POSIX extended ou ERE.

- Cette entête permet d'accéder aussi à un objet ***std::regex()*** qui contient l'expression régulière ainsi que ses options comme ***std::regex_constants::icase*** qui permet d'ignorer la casse.

Les expressions régulières - 2

- **std::regex_match()** : retourne un *booléen* si l'expression match avec la chaîne passée en paramètre.

- **std::regex_search()** : retourne *true* si l'expression est trouvée dans la chaîne passée en paramètre. Permet aussi d'obtenir des informations sur la chaîne trouvée.

- **std::regex_replace()** : effectue une copie de la chaîne et remplace une partie de celle-ci selon l'expression passée en paramètre.

Les expressions régulières : exemple

- Recherche de toutes les occurrences du pattern “sub” dans une chaîne de caractères :

```cpp
#include <iostream>
#include <string>
#include <regex>

int main () {
    std::string str("This subject has a submarine as a subsequence !");
    std::smatch match;
    std::regex expreg("sub[^ ]*");

    std::cout << "Chaine de carateres : " << str << std::endl;
    std::cout << "Regex: /sub[^ ]*/" << std::endl;
    std::cout << "Resultat :" << std::endl;
    while (std::regex_search (str, match, expreg)) {
        std::cout << "  - " << match.str() << std::endl;
        str = match.suffix().str();
    }
    return 0;
}
```

Points essentiels à retenir

- Avec C++ 11, la librairie standard s'enrichit de nombreuses nouvelles fonctionnalités :
  - l'ajout de nouveaux conteneurs,
  - la classe **tuple**,
  - l'utilisation de **pointeurs intelligents** qui facilite la gestion de la mémoire,
  - une amélioration des foncteurs à travers **std::bind**,
  - l'utilisation des threads avec **std::thread**,
  - le support des **expressions régulières**, et bien d'autres choses encore …

IX – Boost et ses principes

Présentation de Boost

- **Boost** est un ensemble de plus de **160 bibliothèques open-source** avec une première version sortie en 1999.

- Elle ajoute de nombreuses fonctionnalités avec une partie de ses bibliothèques qui furent par la suite **intégrées aux différents standards du C++** (C++11, C++14, C++17 et C++20).

- L'objectif de Boost est d'ajouter des **fonctionnalités utiles et qualitatives** au C++. Chacune de ses bibliothèque étant **autonome, indépendante et compatible** avec différents compilateurs C++.

La Pointer Container Library - 1

- La **Pointer Container Library** est une bibliothèque fournissant des conteneurs qui **gèrent automatiquement la mémoire** de leurs objets. Elle est très utile pour stocker des pointeurs vers des objets sans avoir à vous soucier de la gestion de leur mémoire.

- Elle fournit pour cela plusieurs conteneurs qui **stockent des pointeurs** tel que :
  - *ptr_vector, ptr_map, ptr_array, ptr_list, ptr_set, ptr_deque, etc*.

- Elle nécessite l'entête ***<boost/ptr_container/ptr_vector.hpp>***.

La Pointer Container Library - 2

- Dans l'exemple ci-contre, on stocke des objets *Fruit* dans un conteneur *ptr_vector*.

- A la destruction de chacun de ces objets, la mémoire qui leur était allouée est automatiquement libérée.

```cpp
#include <iostream>
#include <boost/ptr_container/ptr_vector.hpp>

class Fruit {
public:
    std::string name;
    Fruit(std::string n) : name(n) { }
    ~Fruit() {
        std::cout << "Destructeur" << std::endl;
    }
};

int main() {
    boost::ptr_vector<Fruit> fruitsVector;

    fruitsVector.push_back(new Fruit("kiwi"));
    fruitsVector.push_back(new Fruit("papaye"));
    fruitsVector.push_back(new Fruit("banane"));
    std::cout << fruitsVector[0].name << std::endl;
    fruitsVector.pop_back();
    return 0;
}
```

La structure de données boost::any

- ***boost::any*** est un objet permettant de stocker des données de n'importe quel type.
- Cet objet provient de la bibliothèque **Boost.any** incluse dans l'entête ***<boost/any.hpp>***.
- ***boost::any_cast*** permet de récupérer la valeur stockée mais si le type demandé ne correspond pas, une exception ***boost::bad_any_cast*** est levée.

```cpp
#include <iostream>
#include <boost/any.hpp>

int main() {
    boost::any value = 42;

    try {
        int intValue = boost::any_cast<int>(value);
        //float floatValue = boost::any_cast<float>(value);
        std::cout << intValue << std::endl;
    } catch (const boost::bad_any_cast& e) {
        std::cerr << "Erreur : " << e.what() << std::endl;
    }
    return 0;
}
```

La structure de données boost::variant

- ***boost::variant*** permet de créer des variables qui vont contenir uniquement les types spécifiées.

- On peut récupérer la valeur avec ***boost::get()*** et si la valeur stockée n'est pas du type spécifiée, une exception ***boost::bad_get*** est levée.

- Entête : ***<bootst/variant.hpp>***

```cpp
#include <iostream>
#include <boost/variant.hpp>

int main() {

    boost::variant<int, std::string> variant("Hello World !");
    variant = 42;

    try {
        if (variant.type() == typeid(int)) {
            int value = boost::get<int>(variant);
            std::cout << value;
        } else if (variant.type() == typeid(std::string)) {
            std::string value = boost::get<std::string>(variant);
            std::cout << value;
        }
        std::cout << " : " << variant.which() << std::endl;
    } catch (const boost::bad_get& e) {
        std::cerr << "Erreur : " << e.what() << std::endl;
    }
    return 0;
}
```

La programmation événementielle

- **Boost** permet de faire de la **programmation événementielle** à travers sa bibliothèque **Boost.Signals2**. Cette bibliothèque offre une implémentation des **signaux**.

- Ce sont des objets qui émettent des **événements** pouvant avoir de multiples cibles. Les signaux sont réceptionnés par des **slots** qui sont ensuite appelés lorsqu'un signal est "émis". Il est même possible de **connecter un signal à un autre signal**.

- Tout cela permet à différents objets de pouvoir communiquer entre eux en fonction de ces évènements. Enfin les signaux sont capables de se **déconnecter automatiquement** des slots lorsque l'un ou l'autre est détruit, et vice-versa.

Connexions et signaux

- Les **signaux** sont créés avec ***boot::signals2::signal()***.
- Le signal est connecté à un **slot** avec la méthode ***connect()***. Les slots peuvent prendre des **paramètres** mais dans ce cas, ils doivent tous prendre les **mêmes types** et le **même nombre** de paramètres.
- Un signal est émis avec **l'opérateur ()** du signal.
- On peut enfin se déconnecter des slots avec la méthode ***disconnect()***.

```
#include <iostream>
#include <boost/signals2.hpp>

void slot1() {
    std::cout << "Slot 1 appelé" << std::endl;
}

void slot2() {
    std::cout << "Slot 2 appelé" << std::endl;
}

int main() {
    boost::signals2::signal<void()> mySignal;

    mySignal.connect(slot1);
    mySignal.connect(slot2);
    mySignal(); // Emet le signal
    mySignal.disconnect(slot1);
    mySignal();
    return 0;
}
```

Gestion des processus - 1

- Boost propose des fonctionnalités de **gestions des processus** à travers la bibliothèque ***boost::process***.
- Cette bibliothèque permet de :
  - chercher des processus,
  - les lancer avec des paramètres,
  - gérer des entrées/sorties synchrone ou asynchrone,
  - grouper des processus ensemble,
  - ou encore manipuler des variables d'environnement.

Gestion des processus - 2

- Ici on cherche le processus firefox dans le **PATH** et on l'exécute dans un **processus enfant**.
- On vérifie s'il tourne avec la méthode ***running()***. Et on attend qu'il se termine avec la méthode ***wait()***.
- On peut aussi exécuter directement un processus avec la méthode ***process()*** qui prend en paramètre **le nom du processus, ses options et des redirections d'I/O**.

```cpp
#include <iostream>
#include <boost/process.hpp>

int main() {
    namespace bp = boost::process;

    try {
        bp::child c(bp::search_path("firefox"));
        if (c.running()) {
            std::cout << "Firefox a été lancé avec succès !" << std::endl;
        } else {
            std::cerr << "Erreur lors du lancement de Firefox." << std::endl;
        }
        c.wait();
    } catch (const std::exception& e) {
        std::cerr << "Exception : " << e.what() << std::endl;
    }
    return 0;
}
```

Mécanisme de communication interprocessus

- La communication interprocessus se fait avec la bibliothèque **boost::interprocess**. Elle permet de :
  - gérer les files d'attentes de messages des processus avec ***boost::interprocess::message_queue***.
  - permettre à plusieurs processus d'accéder à la même zone de mémoire en même temps avec ***boost::interprocess::managed_shared_memory***.
  - mapper des fichiers en mémoire avec ***boost::interprocess::file_mapping***.
  - permettre à plusieurs processus d'accéder à une ressource partagée de manière exclusive ou simultanée avec respectivement ***boost::interprocess::interprocess_mutex*** et ***boost::interprocess::interprocess_semaphore***.

La mémoire partagée - 1

- La gestion de la **mémoire partagée** implique plusieurs étapes :
  - La **création ou l'ouverture** d'un espace de mémoire partagé.

```
using namespace boost::interprocess;
managed_shared_memory segment(open_or_create, "MySharedMemory", 65536);
```

  - L'**allocation** d'un ou plusieurs objets dans cet espace mémoire avec ***construct()***.

```
int *i = segment.construct<int>("MyInteger")(value);
```

  - L'**accès** à cette objet avec la méthode ***find()***.

```
int *i = segment.find<int>("MyInteger").first;
```

La mémoire partagée - 2

- On peut aussi combiner les deux étapes précédentes avec ***find_or_construct()***.

```
int *i = segment.find_or_construct<int>("MyInteger")(value);
```

- La libération de la mémoire partagée, s'il y a besoin, avec ***destroy()***.

```
segment.destroy<int>("MyInteger");
```

- La suppression de l'espace de mémoire partagé avec ***remove()***.

```
shared_memory_object::remove("MySharedMemory");
```

- Entête utilisée : ***<boost/interprocess/managed_shared_memory.hpp>***.

La mémoire partagée - 3

- Exemple complet :

```
#include <boost/interprocess/managed_shared_memory.hpp>

int main() {
    using namespace boost::interprocess;
    int value = 42;
    managed_shared_memory segment(open_or_create, "MySharedMemory", 65536);

    int *i = segment.find_or_construct<int>("MyInteger")(value);
    ++(*i);
    segment.destroy<int>("MyInteger");
    shared_memory_object::remove("MySharedMemory");
    return 0;
}
```

Points essentiels à retenir

- La bibliothèque **Boost** contient de nombreuses bibliothèques permettant de faciliter le travail des développeurs C++.
- La **Pointer Container Librairie** propose des conteneurs qui libèrent automatiquement la mémoire des objets sur lesquels pointent des pointeurs.
- La structure ***boost::any*** permet de stocker n'importe quel type de données.
- La structure ***boost::variant*** stocke des variables de types précis.
- La bibliothèque **Signal2** permet d'envoyer des signaux entre différents objets.
- ***boost::process*** permet d'exécuter et de manipuler des processus.
- ***boost::interprocess::managed_shared_memory*** permet de gérer des zones de mémoires partagées.

X – Bonus

Vérifier la rapidité d'un programme

- Une des manières pour vérifier les performances d'un programme peut être d'utiliser l'objet std::chrono en déclarant une variable de temps début et de temps fin. Une simple soustraction permet ensuite de vérifier la vitesse de différentes instructions.

- Faites toujours plusieurs exécutions afin de vérifier la cohérence des résultats obtenus.

```
auto start = std::chrono::high_resolution_clock::now();
// Instructions ...
auto end = std::chrono::high_resolution_clock::now();
std::chrono::duration<double> elapsed = end - start;
std::cout << "Second duration: " << elapsed.count() << std::endl;
```

XI – Bibliographie

Bibliographie

- « The C++ programming language », 4ème édition par Bjarne Stroustrup.
- *https://en.cppreference.com/w/*
- *https://cplusplus.com/doc/*
- *https://learn.microsoft.com/fr-fr/cpp/cpp/?view=msvc-170*
- *https://devdocs.io/cpp/*
- *https://fr.wikibooks.org/wiki/Programmation_C-C%2B%2B*
- *https://en.wikipedia.org/wiki/C%2B%2B*
- *https://en.wikibooks.org/wiki/C%2B%2B_Programming/Memory_Management*
- *https://www.boost.org/doc/*