# NOTES COMPLÈTES C++ :

# 1. 1. Notes Générales pour 42 :

Compilation :  
- flags : -Wall -Wextra -Werror  
- flag : -std=c++98

Format des classe :  
- UpperCamelCase -> "BrickWall" = BrickWall.hpp

Les interdits :  
- Les mots-clés using namespace <ns\_name> et friend sont interdits  
- Droit à la STL que dans les Modules 08 et 09  
- l’usage des Containers (vector/list/map/etc.) et des Algorithmes (tout ce qui requiert d’inclure <algorithm>) est interdit.

# 2. 2. lib :

#include <iostream>  
- std::cin (entrée standard)  
- std::cout (sortie standard)

#include <string>  
- pour utiliser toute la lib string

#include <iomanip> -> header qui fournit des « manipulateurs » de flux  
- std::setw(n) -> Définit la largeur minimale du champ à n  
- std::setfill(c) -> Caractère de remplissage si la largeur est trop grande  
- std::left / std::right -> Alignement à gauche ou à droite  
- std::setprecision(p) -> Nombre de chiffres significatifs (pour flottants)  
- std::fixed / std::scientific -> Mode d’affichage des flottants

#include <cstdlib>  
- pour utiliser : system("clear") : permet d avoir un interface propre  
- strtod : double strtod( const char \* theString, char \*\* end );   
strtod convertit la chaine en double  
strtod lit autant de caractères numériques qu’il peut et met end sur le premier caractère non reconnu.

#include <fstream> -> permet d'utiliser les ft comme open, close, is\_open  
- utiliser std::ifstream -> input file stream, pour les fichiers / par defaut fais le test ::in  
- utiliser std::ofstream -> output file stream / par defaut fais le test ::out et ::trunc  
- std::ios::in -> open for reading  
- std::ios::out -> open for writing  
- std::ios::trunc -> discard the contents of the stream when opening

#include <cmath>  
- permet d'utiliser les fonctions de la lib math

#include <cstring>  
- permet d'utiliser c\_str qui cast en const char et obtenir l'acces

#include <stdexcept>  
- utilise la class std::exception  
- utilise pour throw, what  
- virtual const char\* what() const throw;

# 3. 3. std :

std::cout <<  
- permet d'ecrire sur la sortie standard (agit comme un printf)

std::cin >>  
- recupere une donne dans l'entree standard, exemple une entree utilisateur

std::string :  
- Une classe de la lib standard (<string>) qui encapsule une suite de caractères de longueur dynamique  
- std::string gère automatiquement la mémoire  
- pas besoin d'allouer ni libérer manuellement un buffer (pas de new[]/delete[])  
- La taille s’adapte : peux stocker aussi bien "Al" que "Alexandre" ou même une phrase plus longue  
- size()/length() -> Nombre de caractères  
- empty() -> Teste si la chaîne est vide  
- operator[]/at() -> Accès direct à un caractère (avec vérification via at)  
- append(), + -> Concaténation  
- substr(pos, len) -> Extraction d’une sous-chaîne  
- find(str) -> Recherche d’une sous-chaîne  
- compare() -> Comparaison lexicographique  
- c\_str() -> Conversion en C-string (const char\*)  
- getline -> stocke dans une chaîne de caractères (std::string ou un tableau de char, selon la version utilisée), std::getline(std::cin, ligne)

std::ifstream  
- utiliser pour l'ouverture de fichier infile

std::ofstream  
- utiliser pour creer un fichier de sortie outfile

std::cin.eof()  
- Après une opération de lecture qui tombe sur la fin du fichier (ou sur Ctrl-D / Ctrl-Z en terminal)  
- le drapeau EOF est levé et std::cin.eof() renvoie true.

# 4. 4. fonction membre :

- variable dans la class et utiliser avec une instance afin que le class soit utile

# 5. 5. this :

- En C++, this est un pointeur implicite disponible dans toutes les fonctions membres d'une classe.

- Il pointe vers l'instance actuelle de l'objet sur laquelle la méthode est appelée.

- utile pour : retourner l'objet courant (return \*this)

- utile quand : un paramètre ou une variable locale a le même nom qu’un attribut membre (this->foo = foo)

Utilisation :

- this->foo -> Accède à l’attribut foo de cet objet

- this->bar() -> Appelle la méthode bar() de cet objet

- \*this -> Donne l’objet lui-même (en déréférençant le pointeur this)

- return \*this -> Souvent utilisé pour permettre le chaining (enchaînement de méthodes)

# 6. 6. initialization list :

Initialiser les 3 parametres en list :

Sample::Sample(char p1, int p2, float p3) : a1(p1), a2(p2), a3(p3)

- syntaxe optimal

- assigne la valeur de p1 dans a1

# 7. 7. visibility :

public et prive peuvent etre utiliser dans les fichiers autre que main

public :  
- Membres public et protected restent accessibles  
- possible d acceder depuis l interieur et exterieur de la class

protected :  
- Membres public deviennent protected

private :  
- Membres public/protected deviennent private  
- possible d acceder depuis l interieur de la class  
- Pour masquer l’héritage à l’extérieur

# 8. 8. accessor c++ :

Les accesseurs sont des méthodes publiques qui permettent de lire (get) ou modifier (set) des attributs privés d’une classe

Cela respecte le principe d'encapsulation : protéger les données et contrôler leur accès

Utiliser get et set permet de :  
- Protéger les données de l'objet  
- Valider ou filtrer les entrées avant modification  
- Rendre le code plus fiable et maintenable

get :  
- Lire un attribut, retourne une copie (return this->\_variable)  
- const std::string& getFirstName()const;

const std::string& Contact::getFirstName()const  
{  
 return this->\_firstName;  
}

set :  
- Modifier un attribut (this->\_variable)  
- Peut inclure des vérifications  
- void setFirstName(const std::string& firstName);

void Contact::setFirstName(const std::string& firstName)  
{

this->\_firstName = firstName;  
}

# 9. 9. class Canonical form :

class :  
- une class par defaut a un scope prive  
- Constructeur par défaut  
- Constructeur de recopie  
- Opérateur d’affectation  
- Destructeur

class Sample{  
 public:  
 Sample(void);  
 Sample(Sample const& src);  
 Sample& operator=(Sample const& other);  
 ~Sample(void);  
};

std::ostream& operator<<(std::ostream& other, Sample const& Sample);  
———-  
Sample(Sample const& src);  
- Copie tout le contenu de l’objet src dans l’objet courant (this)

Sample& operator=(Sample const& other);  
- C’est un appel à l’opérateur d’affectation (operator=)  
- exemple a = b, copie les donnee de b dans a  
- assigner une autre instance a partir de cette class  
- But : copier l’état (les données internes) d’un objet other dans un autre objet déjà existant (\*this)

std::ostream& operator<<(std::ostream& other, Sample const& Sample)  
- But : définir comment un objet Sample est affiché dans un flux de sortie (std::cout, fichier, etc.).  
- permet de chainer les arguments  
- evite d utiliser get a chaque appel  
exemple :  
std::ostream& operator<<(std::ostream& os, Bureaucrat const& Bureaucrat){  
std::cout << Bureaucrat.getName() << Bureaucrat.getGrade() << std::endl;  
return os;

# 10. 10. Operator :

Operation arythmetique :  
1 + 1 = operateur infixe  
+ 1 1 = operateur prefixe  
1 1 + = operateur postfixe

4 opérateurs de arithmétiques : +, -, \*, et /  
- <class> operator+(const <class> &other);  
- <class> operator-(const <class> &other);  
- <class> operator\*(const <class> &other);  
- <class> operator/(const <class> &other);

4 opérateurs d’incrémentation et de décrémentation   
“pré-incrémentation et post-incrémentation”  
“ pré-décrémentation et post-décrémentation”

- <class>& operator++();  
- <class> operator++(int);  
- <class>& operator--();  
- <class> operator--(int);

pre incremente, +1 a la valeur, return l adress avec \*this (++valeur)  
{  
 ++this->\_value;  
 return \*this;

}

post incremente, return ancienne valeur et ensuite la valeur modifier (valeur++)  
{  
 <instance\_class> old = \*this;  
 operator++();  
 return old;  
}  
  
- Meme chose pour le ‘-‘

6 opérateur de comparaison : >, <, >=, <=, == et !=  
- bool operator>(const <class> &other)  
- bool operator<(const <class> &other)  
- bool operator>=(const <class> &other)  
- bool operator<=(const <class> &other)  
- bool operator==(const <class> &other)  
- bool operator!=(const <class> &other)

# 11. 11. les pointeurs :

std::string str = "HI THIS IS BRAIN"  
- declarer la variable

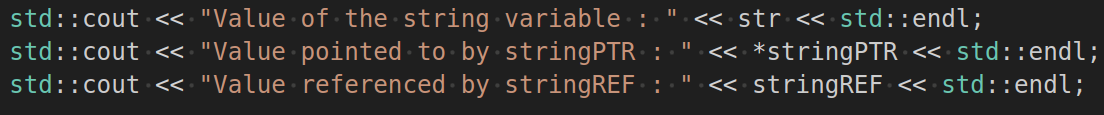
std::string \*stringPTR = &str  
- comme en c, c'est un pointeur vers la variable str  
- peut etre null

std::string &stringREF = str  
- référence à la variable string  
- Elle agit comme un alias, un autre nom pour la même variable  
- stringREF est exactement la même chose que string, juste un autre nom pour y accéder  
- utiliser dans le cas ou la variable ne peut pas etre nul (exemple, un homme toujours equiper d'un objet)

- std::string str = "HI THIS IS BRAIN"; -> a une adresse memoire

- std::string \*stringPTR = &str -> &str récupère l’adresse mémoire de la variable str

- std::string &stringREF = str -> &stringREF devient une référence à str (même adresse, même contenu)  
  
exemple affichage :



# 12. 12. filestreams :

#include <fstream>

Types :  
- std::ifstream -> lecture de fichier (input)  
- std::ofstream -> écriture dans un fichier (output)

Modes d’ouverture (std::ios flags) :  
- std::ios::in -> Lecture (par défaut pour ifstream)  
- std::ios::out -> Écriture (par défaut pour ofstream)  
- std::ios::trunc -> Vide le fichier à l'ouverture

Méthode Description  
- .open() -> Ouvre un fichier (si pas ouvert via constructeur)  
- .close() -> Ferme le fichier  
- .is\_open() -> Vérifie si le fichier est bien ouvert  
- operator>> -> Lecture comme avec cin  
- operator<< -> Écriture comme avec cout

Autres points clés :  
- c\_str() : convertit un std::string en const char\* (utile pour .open()).  
- Lors de l'écriture, std::ofstream écrase le contenu par défaut (trunc).  
- Toujours vérifier si un fichier est ouvert avant de lire/écrire.  
- exemple ifs.is\_open(); ou if (!ifs) suffit

# 13. 13. Ad-hoc Polymorphisme :

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

void bar(char const c)const;  
 void bar(int const n)const;  
 void bar(float const z)const;  
 void bar(Sample const & i)const;  
};

- 4 surcharges de la fonction membre bar  
- Meme nom de fonction mais avec des parametres differents  
- Permet d'utiliser seulement la fonction bar afin d'afficher les parametres differents

# 14. 14. allocation en c++ :

Sur la pile (stack) :  
- Objet déclaré localement dans une fonction ou une portée.  
- Destruction automatique à la fin du scope.  
- Avantages : rapide, sûre (pas de fuite mémoire).  
- Limite : portée restreinte, taille fixe à la compilation.

Sur le tas (heap) :  
- Objet alloué via new (ou new[] pour les tableaux).  
- Reste en mémoire jusqu’à un delete explicite.  
- Avantages : plus flexible, persiste au-delà d’un scope.  
- Inconvénient : gestion manuelle (risque de fuite mémoire si delete oublié).

Construction / Destruction  
new :  
- Alloue de la mémoire sur le tas.  
- Appelle le constructeur de l’objet.

delete :  
- Libère la mémoire allouée par new.  
- Appelle le destructeur de l’objet.

Règles :  
- Chaque new → un delete  
- Chaque new[] → un delete[]

Exemple d'utilisation :  
Zombie\* z = new Zombie("Alice"); // Allocation dynamique  
z->announce(); //fonction d'affichage  
delete z; // Libération manuelle

Allocation tableaux de zombies :  
Zombie\* horde = new Zombie[N]; // Crée un tableau de N zombies

**Exemple pour eviter les leak :**  
⃕

void testShrubbery()  
{  
 std::cout << "TEST 1" << std::endl;  
 AForm \*sch = NULL; -> declare instance en dehors du try, si erreur delete de suite

try  
 {  
 Bureaucrat bureaucrat("david", 130);  
 sch = new ShrubberyCreationForm("Garden");  
 std::cout << \*sch << std::endl;  
 bureaucrat.signForm(\*sch);  
 bureaucrat.executeForm(\*sch);  
 }  
 catch(std::exception& e)  
 {  
 std::cerr << "Erreur : " << e.what() << std::endl;  
 }  
 delete sch; -> permet de placer le delete en dehors  
 std::cout << std::endl;  
}

# 15. 15. Heritage c++ :

Types d’héritage :

public :  
- Membres public et protected restent accessibles

protected :  
- Membres public deviennent protected

private :  
- Membres public/protected deviennent private  
- Pour masquer l’héritage à l’extérieur

Constructeurs et destructeurs :  
Construction :  
- Appel du constructeur de la classe de base  
- Puis appel du constructeur de la classe dérivée

Destruction :  
- Appel du destructeur de la classe dérivée  
- Puis appel du destructeur de la classe de base  
- Cela garantit que tout est construit et détruit proprement, du plus général (base) au plus spécifique (dérivé).

Appels de constructeurs (héritage) :

class ScavTrap : public ClapTrap  
- class derive ScavTrap qui herite de la class de base ClapTrap

Constructeur classique .cpp :  
- ScavTrap::ScavTrap(std::string name) : ClapTrap(name)

Constructeur de copie .cpp :  
- ScavTrap::ScavTrap(const ScavTrap& src) : ClapTrap(src)  
- copie spécifique à ScavTrap

Surcharge de l’opérateur d’affectation .cpp :  
ScavTrap& ScavTrap::operator=(const ScavTrap& other)  
{  
 if (this != &other)  
 {  
 ClapTrap::operator=(other);  
 }  
return \*this;  
}

- Copie les attributs de ClapTrap  
- Copier les attributs spécifiques à ScavTrap s’il y en a  
- si une variable est en plus il faut la copier, si une class a un nom on rajoute la copie du nom

la class derive possède tout ce que la class de base possède :  
- Attributs : variable etc.  
- fonction membre  
- surcharger ou redéfinir des méthodes héritées (ex : une fonction())

dans la class ClapTrap - Pourquoi protected au lieu de private ?  
private :  
- inaccessible depuis les classes dérivées.

protected :  
- accessible uniquement par les classes qui héritent, mais pas à l’extérieur.

# 16. 16. Heritage en diamant c++ :

ClapTrap

/ \

ScavTrap FragTrap

\ /

DiamondTrap

Problème du diamant :

DiamondTrap hérite deux fois de ClapTrap :  
- une fois via ScavTrap  
- une autre via FragTrap  
- Donc il y a deux copies de ClapTrap dans DiamondTrap.  
- Si on écrit DiamondTrap::name, le compilateur ne sait pas laquelle utiliser : ScavTrap::ClapTrap::name ou FragTrap::ClapTrap::name.  
- Solution : l’héritage virtuel  
- L’héritage virtuel garantit qu’il n’existe qu’une unique instance de la classe de base, même si plusieurs chemins d’héritage y mènent

En résumé :

Sans virtual  
- Deux copies de la classe de base  
- Accès ambigu aux membres hérités  
- Erreurs à la compilation

Avec héritage virtuel  
- Une seule copie de ClapTrap  
- Plus clair et propre  
- Résolu grâce au virtual

# 17. 17. Sub-typing polymorphism :

- Le polymorphisme de sous-typage en C++ permet de traiter des objets de classes dérivées via des pointeurs (ou références) vers leur classe de base et d’appeler la bonne version d’une méthode à l’exécution grâce aux fonctions virtuelles

Principe :  
  
Sans virtual :   
- l’appel d’une méthode via un pointeur vers la classe de base utilise toujours l’implémentation de la base (liaison statique à la compilation)

Avec virtual :   
- l’appel est résolu selon le type réel de l’objet pointé (liaison dynamique à l’exécution)

Exemple :  
- class principal Animal -> un animal peut emettre un son  
- class derive Cat -> le chat emet un son (par exemple il miaule)  
- avec virtual sur la fonction membre on va dire que le chat miaule  
- sans virtual le chat executera le print qui est dans la class principale (le chat emet un son)

exemple fonction membre avec virtual :

- virtual void makeSound()const;

Ne pas oublier de mettre virtual au destructeur

- virtual ~Animal();

- on met virtual dans le destructeur de la classe de base pour assurer que la destruction des objets dérivés via un pointeur de base se fasse entièrement et en toute sécurité

- En marquant virtual ~Base(), tu dis au compilateur : « Pour ce destructeur, choisis la version à l’exécution en fonction du vrai type de l’objet (Derived ou autre) »

Exemple concret :

- Base\* p = new Derived;

- p = class de base

- Derived = class derive

- p->f(); // appelle Derived::f() si f() est virtuelle

- sinon appelle Base::f() si f() est pas virtuelle

# 18. 18. Abstract classes and interfaces :

Utiliser des methodes pure

exemple :   
virtual void attack(string) = 0;

class ACharacter  
{

//A pour abstrait

}

Consequence :  
- on ne peut l'implementer car definit a 0  
- on ne va pas pouvoir instancier la class (elle sera donc abstraite)

A quoi ca sert :  
- certain comportement sont defini tandis que d'autre non

exemple concret :  
- on a une class Character avec une methode pure (class mere)  
- virtual void attack(string) = 0;  
- on a une class warrior qui va heriter de character avec la fonction membre attack  
- virtual void attack(string);  
- en soit la class warrior devra heriter de la fonction attack de la class mere (meme si la class n'en a pas) car un character ne peut pas attaquer  
- un character n'existe pas en soit, c'est un warrior, un mage etc

# 19. 19. class imbriquees et Exceptions :

Header :  
- include <stdexcept>  
- include <exception>

Une exception est un maniere de remonter un message a travers la pile d'appel lorsque l'on trouve une erreur  
- cela evite de faire des return -1, -2 etc comme en C  
————  
virtual const char\* what() const throw():  
1.virtual  
- Cela veut dire que cette fonction peut être redéfinie (surchargée) dans une classe dérivée.  
- Exemple : si vous créez une exception personnalisée, vous pouvez donner votre propre version de what().

2.const char\*  
- Le type de retour est un pointeur vers une chaîne de caractères en style C (terminée par '\0').  
- Cette chaîne décrit généralement le message d’erreur.

3.what()  
- Le nom de la fonction. Pas de paramètres.  
- Sert à retourner une description lisible de l’exception.  
- const (après la parenthèse) Indique que cette méthode ne modifie pas l’objet courant.  
- Donc vous pouvez appeler what() sur un objet constant.

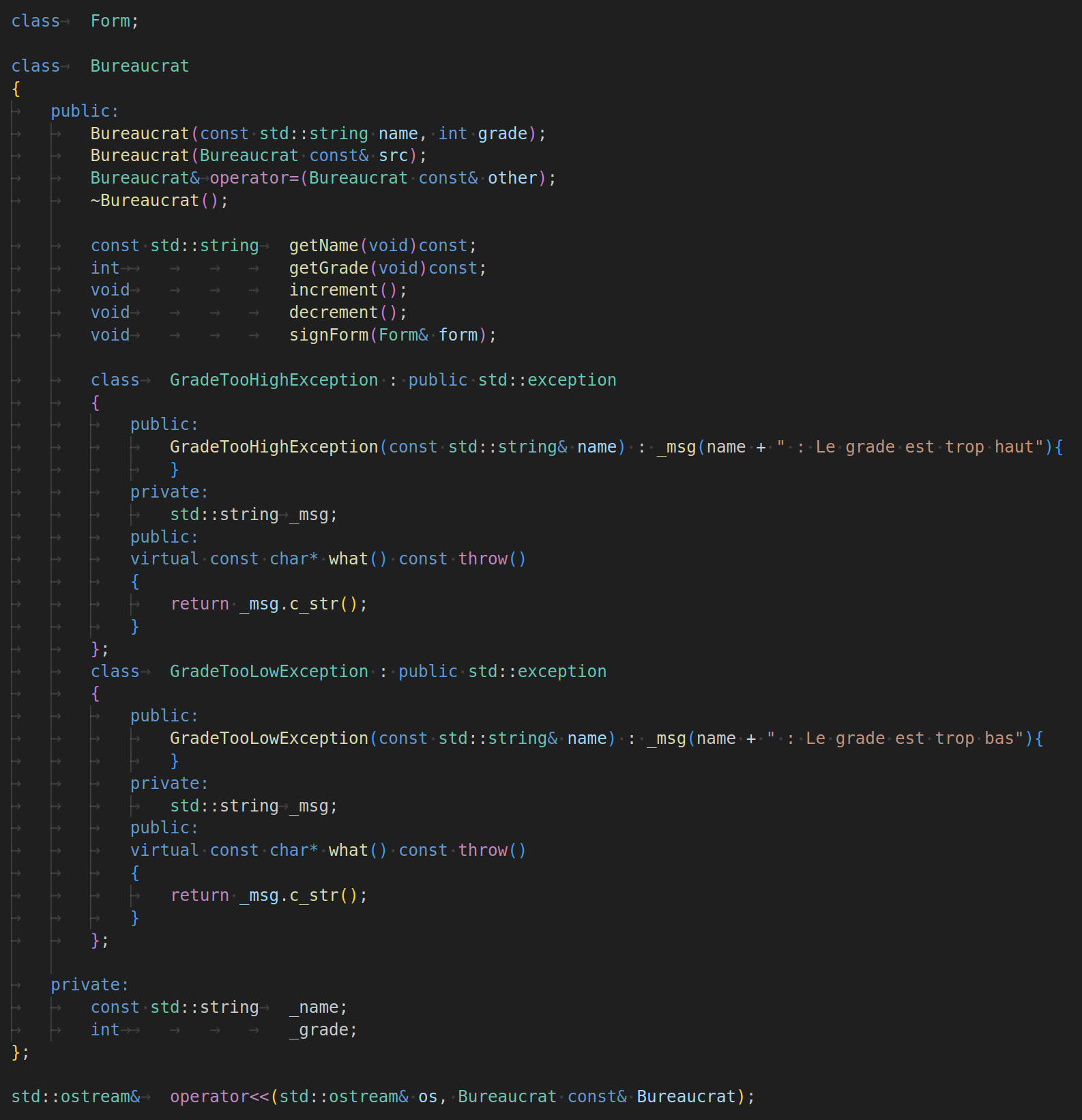
4.throw()  
- on peut balancer une exception depuis ce throw

throw :  
- Je balance une erreur  
- sert à lancer une exception (c’est-à-dire signaler une erreur).  
- Quand tu fais throw, tu arrêtes le cours normal du programme et tu envoies une "erreur attrapable" par un try/catch.

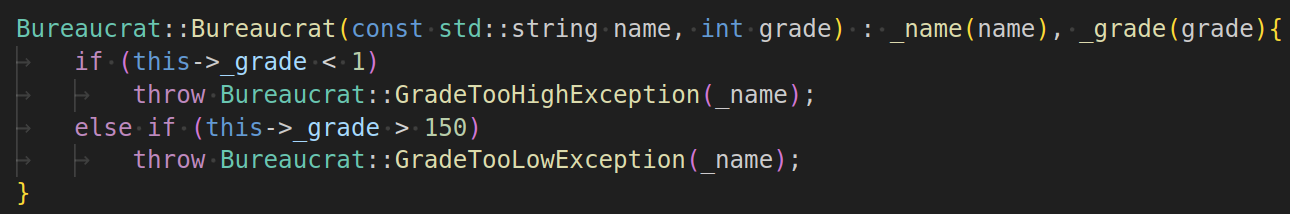
catch :  
- J’attrape l’erreur

what() :  
- Elle renvoie un message d’erreur en texte (de type const char\*) qui décrit ce qui s’est passé.  
- est une fonction fournie par la classe de base std::exception.

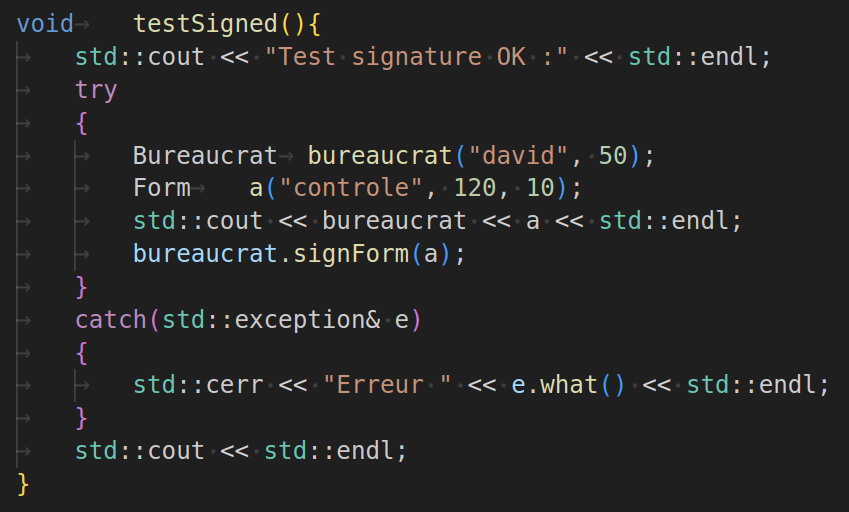
Exemple de class imbriquee avec exception :



1. class de base  
2. forme canonique (constructeur, copie etc)  
3. fonction membre et getter  
4. class imbriquee pour gerer l'exception 1  
5. en public, un constructeur pour genere un message  
6. en public, une méthode membre virtuelle d’une classe (virtual const char\* what() const throw()), permet de gerer le return du message (c\_str cast en const char)  
7. class imbriquee pour gerer l'exception 2  
8. pareil que pour la premiere class imbriquee  
9. operateur d'insertion afin d'afficher une instance comme souhaite  
Exemple throw :



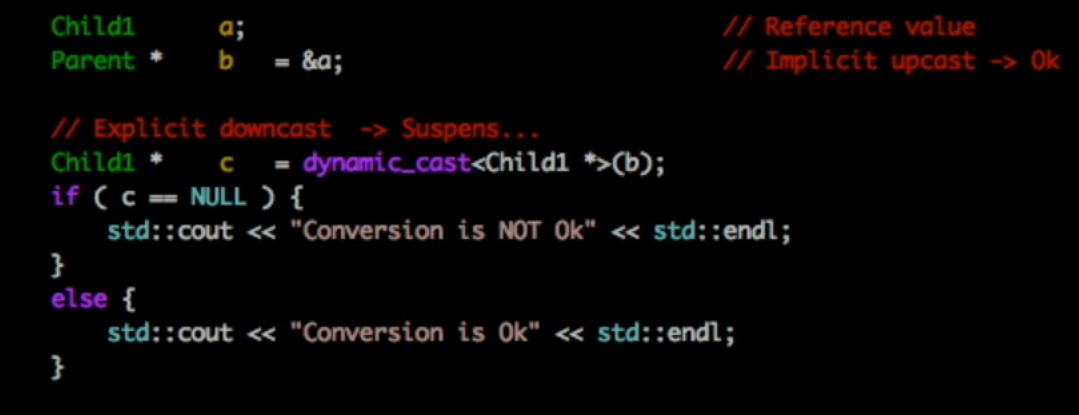
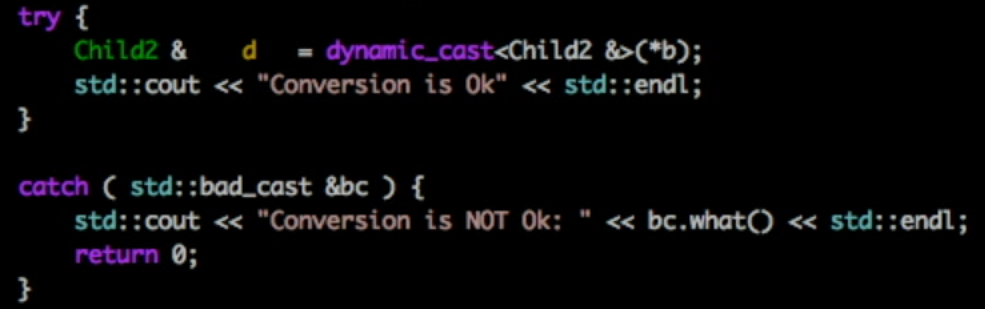
Exemple try, catch and what :



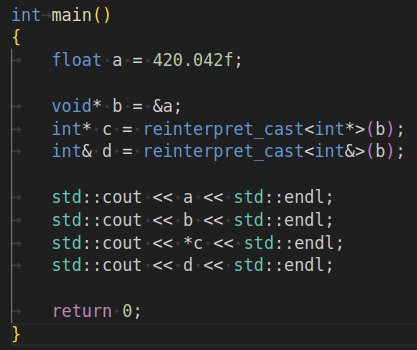
1. j "essaie" de creer les instances avec try  
2. si une erreur est trouvee rentre dans le catch  
3. remonte l'erreur et affiche avec what

# 20. 20. cast c++ :

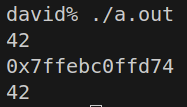
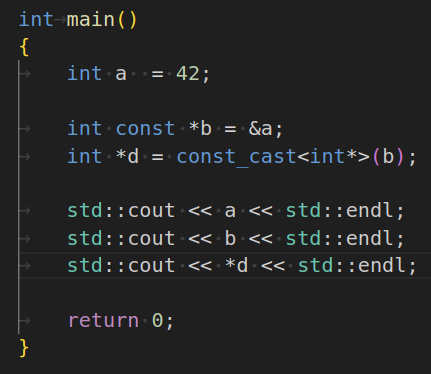
**static\_cast : (se fait a la compilation) -> static\_cast<>()**int main()  
{  
 double b = 42.0;  
 int d = static\_cast<int>(b)  
}  
- on vient cast b en int, d contient 42  
- permet de faire des convertion simple  
  
—————  
  
**dynamic\_cast :** **(tres interessant, se fait a l’execution) -> dynamic\_cast<>()**- ne fonctionne que dans le cas de polymorphisme par sous typage  
- au moins avoir une methode virtual pour utiliser le dynamic  
- fonctionne uniquement sur des cast de pointeur ou de reference  
- va renvoyer soit NULL si le cast n est pas possible (pour les pointeurs)  
- si cast sur un pointeur va renvoyer l’adresse  
- dans le cas d’ une reference, on ne renvoie pas NULL, une reference ne peut pas etre NULL  
- pour une reference il faut lever une exception (std::bad\_cast), try and catch  
  
cast sur pointeur :  
  
  
  
  
  
  
  
  
  
  
  
  
  
cast sur une reference :



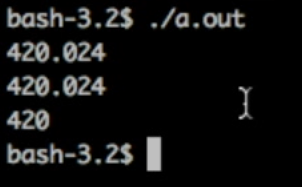
**reinterpret\_cast : (le plus permissif) -> reinterpret\_cat<>()**- aucune verification pour la compilation (l’execution peut poser probleme)  
- utile pour les retypages, permet de retype un void\* sous la forme d un char\* par exemple



**const\_cast :**



**Cast operators :**- va permettre de definir au sein de nos class des operateur specifique afin de pouvoir faire des convertion implicite de notre class vers un type qui nous interesse

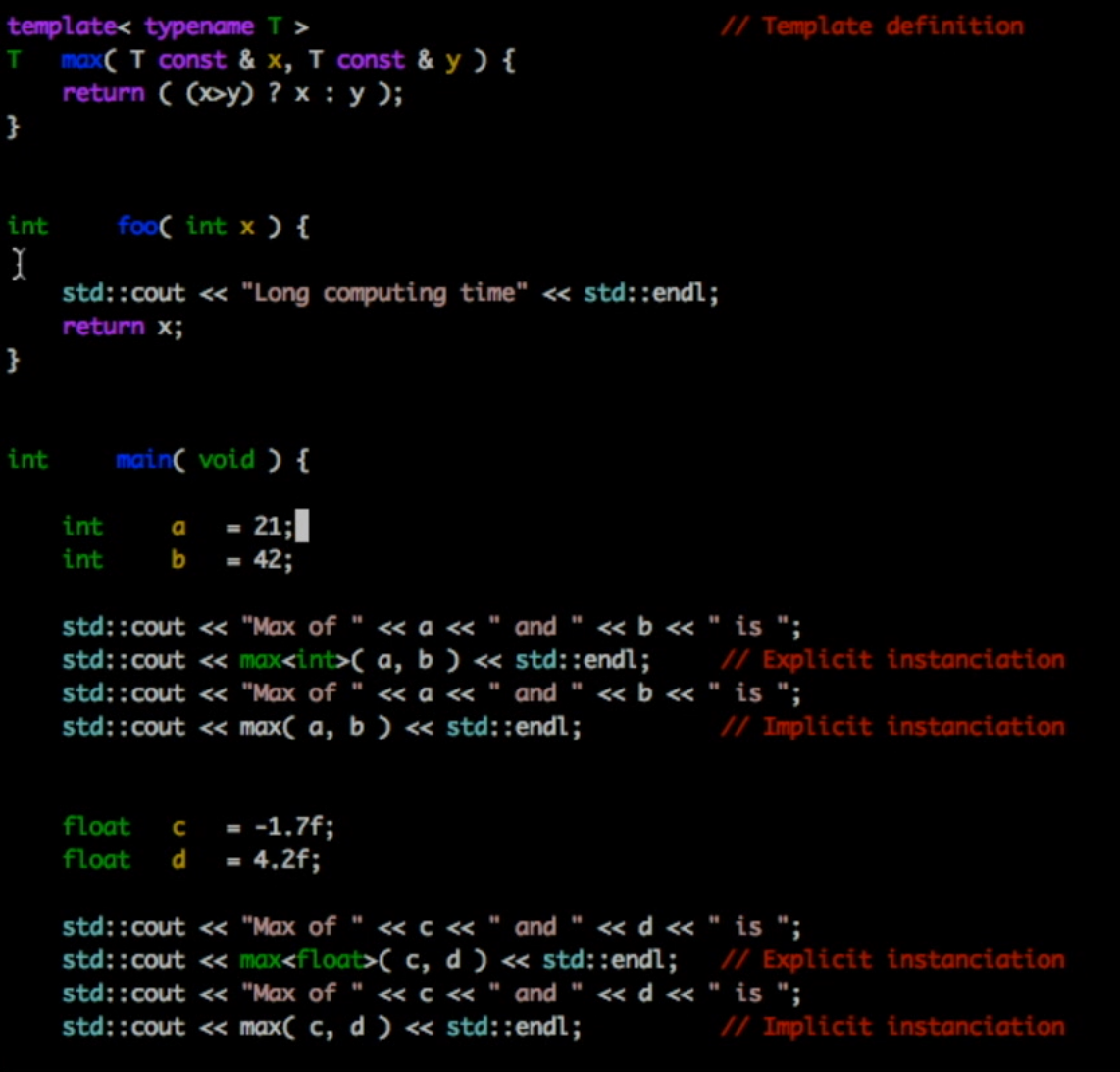


f

vdgdfgd

21. 21. Templates c++ :

- Un template est un patron de code, un modele  
  
- Pour indiquer que c est un template :  
template <typename T>  
<typename T> -> le T est une variable comme int, float etc  
  
- ‘T’ designe un type quelconque mais toujours le meme



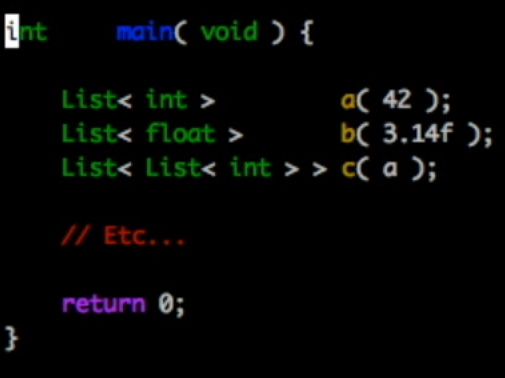
2 moyens de faire une instancation de note template :  
  
**explicit :**  
- appeler la fonction et mettre entre chevron le type sur lequel on veut instancier notre template et passerles types a la suite entre () -> std::cout << max<int>(a, b)  
  
**implicit :**- moins utiliser surtout pour des gros codes -> std::cout << max(a, b)

Definition dans vscode : <nom class>.class.tpp (pas obligatoire)

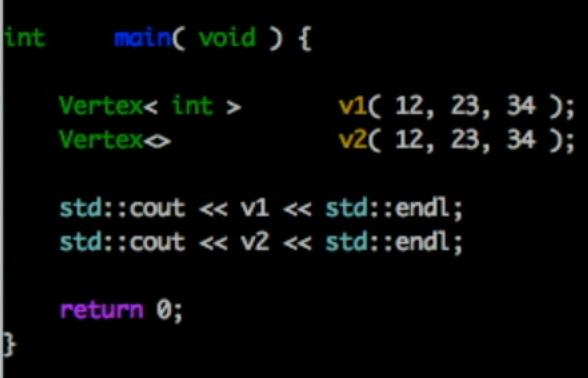
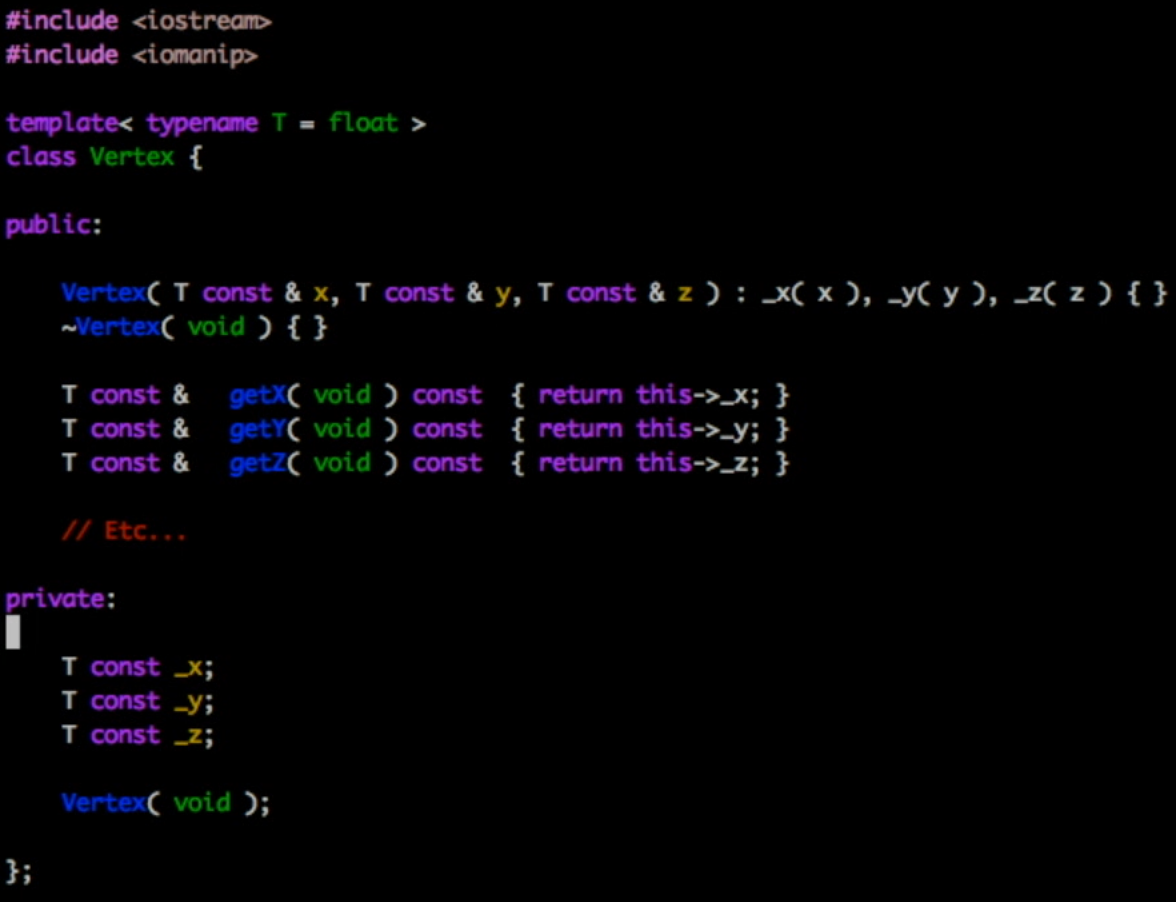
Class et stuct template :



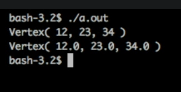
Utilisation main :



# **Default type :** template<typename T = float>



Vertex<> v2(12, 23, 34);  
- si on ne precise pas le type ce sera le type par defaut definis dans la class comme ci dessus  
**template<typename T = float>** -> type par defaut  
- les nbr seront converti en float



**Specialization :**

# 

# SOUS CHAPITRE : 22. 1.1 représentation des nombres en virgule fixe :

static const int bits = 8

- division entière par 256

exemple concret : représenter la valeur 5,75 en fixed-point

- \_value = 5.75 × 256 = 1472 -> premiere chose multiplier sa value par 256

toInt : récupère uniquement la partie entière (tronquée)

Calcul pour \_value = 1472 :

int Fixed::toInt(void) const {

return this->\_value >> Fixed::bits;

}

- Opération : décalage à droite de 8 bits équivaut à une division entière par 256

- 1472 >> 8 ≡ 1472 / 256 = 5.75 (division entière)

- resultat = 5 (la partie fractionnaire 0.75 est tronquée)

toFloat : restitue la valeur décimale exacte stockée

Calcul pour \_value = 1472 :

float Fixed::toFloat(void) const {

return (float)(this->\_value) / (1 << bits);

}

- (float)1472 / 256.0f = 5.75

- \_value ne contient jamais le « vrai » nombre à la sortie, il ne contient que la valeur entière échellée

- si toInt -> value = 5

- si toFloat -> value = 5.75

exemple concret :

/\*

a = 0

print 1 : 0

pre-incremente : 0 devient 1 -> 1/256

print 2 : 0.00390625

print 3 : 0.00390625

post-incremente : 1 devient 2, affiche ancienne valeur

print 4 : 0.00390625

print 5 : 0.0078125 nouvelle valeur, 2/256

b : 5.05 \* 2

5.05 \* 256 = 1292.8 roundf 1293

2 \* 256 = 512

1293 \* 512 = 662016

662016 / 256 = 2586

2586 / 256 = 10.10

print 6 : valeur de b = 10.1016

print 7 : valeur max entre a et b

\*/

- 2 divison car 2 multiplication

- autre cas, une seul division

Multiplication (operator\*) fait :

produit 64 bits : 1293 × 512 = 662016

décale à droite de 8 bits : 662016 >> 8 = 662016 / 256 = 2586

construit Fixed(2586)

Valeur de b = 2586 / 256 = 10.1015625 → affiché 10.1016

# 22. 1.2 pseudo\_literal

En C++98 certains littéraux spéciaux sont reconnus comme des représentations de valeurs flottantes particulières.  
  
nan / nanf :  
- Not A Number, résultat indéfini d’une opération comme 0.0/0.0 ou sqrt(-1.0)  
  
+inf, -inf, +inff, -inff :   
- les valeurs flottantes infinies positives ou négatives.  
  
La différence entre ceux qui finissent par f et ceux sans f :  
- Avec f -> constante de type float (nanf, inff)  
- Sans f -> constante de type double (nan, inf)  
  
Il n’existe pas de pseudo-littéraux pour int ou char, car ces types ne peuvent pas représenter l’infini ni NaN  
  
float :   
- peut contenir nanf, +inff, -inff  
  
double :   
- peut contenir nan, +inf, -inf  
  
int :   
- ne peut pas représenter ces cas (impossible)  
  
char :   
- idem que pour int, impossible de représenter NaN ou inf.