TD/TP 10

Exceptions & flots

Objectifs, méthodes

Les **exceptions** sont un mécanisme de rupture de la séquence d'exécution normale vers un code de traitement d'erreurs (bloc **catch**) dès qu'une anomalie est signalée (exception lancée : **throw**) soit par notre code soit par le code des bibliothèques utilisées. Le mécanisme ne marche que si l'exécution se fait dans ou depuis un bloc **try**. devant le bloc catch.

Les **flots** permettent de transformer des données complexes de différents types (entiers, flottants, chaînes, structures composites...) en données séquentielles de types flots d'octets (**stream**) et réciproquement. Un stream peut être vers/depuis une chaîne ou vers/depuis un fichier ou vers/depuis la console. Ces flots sont un outil indispensable de communication entre le logiciel et le monde extérieur : les entrées/sorties.

Exceptions et flots n'ont pas de lien direct, mais ils ont été présentés dans le même cours dans la mesure où la gestion des anomalies devient un problème inévitable dès qu'on s'intéresse aux entrées/sorties : le logiciel n'a pas le même contrôle sur le monde extérieur que sur son petit monde intérieur.

L'objectif est de pratiquer ces mécanismes nouveaux (cin/cout ne sont pas nouveaux mais sont maintenant considérés comme des instances particulières de istream/ostream). Ce sujet essaye de proposer des exercices simples et progressifs pour tous, ponctués de corrigés pour ne pas rester bloqué ou pour confirmer votre approche ou si vous voulez voir une façon « canonique » de faire.

Les concepts du cours couverts par ce TP seront (pas nécessairement dans cet ordre)

- **`exceptions`**
- **`flots fichiers`**
- **`flots chaînes`**
- **`sérialisation`** (inévitablement long : facultatif)

1. Le caractère `'\n'`, les chaînes multi-lignes, `std::ostringstream`

D'abord dissipons un malentendu qui risque de brouiller la compréhension des exos suivants : nous sommes bien d'accord que quand on parle de « une chaîne de caractère » ça n'est pas la même chose que « une phrase sur une seule ligne ». Une chaîne est une séquence d'octets, parmi les octets possibles il y a le `'a'` (ASCII 97) le `'A'` (ASCII 65) le `'@'` (ASCII 64) etc... et le `'\n'` (ASCII 10[1])

Le « retour ligne » est donc un caractère comme un autre, il se **stocke** très bien au milieu/à la suite des autres dans la séquence des octets d'**une chaîne**. Sa spécificité n'apparaît que lors d'un **affichage** : le caractère `'\n'` quand il est affiché est « invisible » mais fait passer la suite à la ligne.

1 En fait le `'\n'` a une correspondance ASCII qui dépend du système. Pour des raisons à la fois historiques et d'absence de bonne volonté (abus de position dominante) la console et les fichiers Windows utilisent un doublet d'octets (ASCII 13-ASCII 10) ce qui est une source constante de friction et d'incompatibilité. En interne dans nos programme le `'\n'` est bien représenté par un seul octet de code ascii 10, il tient bien sur un seul char.

Et du côté des entrées, quand il apparaît dans une saisie il signale la validation de la saisie, il est donc difficile de saisir un `'\n'` dans un char ou une chaîne de la même façon que les autres. On verra qu'en C++ les espaces en général jouent le même rôle de séparateur (mais pas de validateur !)

Je vous propose quelques codes de tests pour s'assurer qu'il n'y a aucune confusion par rapport au rôle de `'\n'` :

- spécifique dans les saisies/affichages : ce n'est pas un caractère comme un autre
- pas du tout spécifique dans les chaînes : c'est un caractère comme un autre

Créez un nouveau projet C++, appelez le ***`tests_retours_lignes`*** *par exemple...*

D'abord testons que je ne raconte pas de bêtises. Pour afficher le code ASCII et non pas le symbole d'un caractère on peut le *caster* en type entier. *Vérifier les codes indiqués page précédente* pour le `'a'` le `'A'` le `'@'` et le `'\n'` (des fois il faut le voir pour le croire) :

```
std::cout << (int)'a' << std::endl;
```

De ce point de vue `'\n'` est comme un autre. Mais si on essaye de le saisir on a une surprise, dans ce code *essayer successivement* de saisir le `'a'` le `'A'` le `'@'` et le retour ligne et l'espace :

```
char c;
std::cin >> c;
std::cout << (int)c << std::endl;
```

Retour ligne et espace sont des séparateurs, ils sont donc ignorés dans les entrées en tant que valeurs à saisir, ils ne peuvent pas être saisis, pas directement... Si on veut **capter** les espaces dans une saisie, il faut saisir une **phrase** dans une chaîne. Ce qui implique (bizarrement, je suis d'accord) d'abandonner la syntaxe >> et d'utiliser la fonction **getline**. *Tester le code ci-dessous à gauche* avec une phrase comme « Bonjour le monde ! », puis avec une phrase qui commence par des espaces « Bonjour le monde ! ». Voyez que les espaces n'entrent par avec >> ils servent de séparateurs mais ils sont ignorés en tant que valeur. De même avec le code à gauche on ne peut pas entrer une phrase vide, tester. *Tester maintenant le code ci-dessous à droite* avec les mêmes phrases, vérifier que le fait d'entrer une phrase vide (valider directement) conduit non pas à une chaîne avec \n dedans mais bien à une chaîne vide. Le \n de validation est enlevé de la chaîne saisie par getline !

```
std::string ligne;
std::cin >> ligne;
std::cout << "\"" << ligne << "\"" << std::endl;
```

```
std::string ligne;
std::getline(std::cin, ligne);
std::cout << "\"" << ligne << "\"" << std::endl;
```

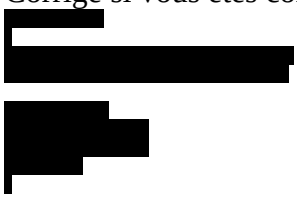
Revenons un instant sur \n dans une chaîne. J'ai dit qu'on pouvait avoir un \n dans une chaîne comme un caractère normal. *Afficher depuis le main la chaîne retournée par cette fonction* :

```
std::string foo()
{
    return "Ligne 1 patati patata\nLigne 2 yodli yodla\n";
}
```

Pourrait-on remplacer directement les \n de cette chaîne constante par des std::endl ? Pourquoi ? Et si on voulait assembler cette chaîne ligne pas ligne ? On peut utiliser la **concaténation**, partir d'une chaîne vide et ajouter les lignes une par une : *compléter* la version 2 de foo pour obtenir le même résultat que précédemment depuis l'appel du main.

```
std::string foo() // correction page suivante...
{
     std::string ligne1 = "Ligne 1 patati patata";// utiliser, pas modifier
     std::string ligne2 = "Ligne 2 yodli yodla";  // utiliser, pas modifier
     ... à compléter !
```

Corrigé si vous êtes coincé (sélectionner, copier, puis coller dans votre code) :



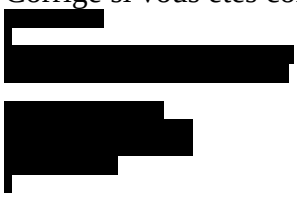
La concaténation marche pour construire une chaîne complexe, y compris une chaîne multi-lignes, à partir d'informations en morceaux. Mais en plus de nécessiter une syntaxe spécifique elle présente l'inconvénient de ne pas être aussi souple qu'un affichage direct avec std::cout<<. En particulier il serait plus difficile d'y injecter des valeurs numériques. Il peut donc être intéressant de **traiter une chaîne comme std::cout**. Ce n'est pas possible directement dans une std::string mais c'est possible avec un objet std::ostringstream, lequel peut ensuite facilement se convertir en string :

```
std::string tableDeSept()
{
    std::ostringstream oss;  // #include <sstream>

    oss << "Table de 7" << std::endl;
    for (int i=1; i<=10; ++i)
        oss << "7 x " << i << " = " << 7*i << std::endl;

    return oss.str();  // renvoie la chaîne ci contre =>
}
```

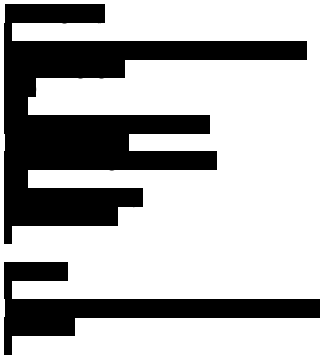
```
Table de 7
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70
```

Sur le même principe, *modifier le code de foo pour assembler la chaîne résultante avec un ostringstream.* Utiliser std::endl à la place de \n. Le résultat, vu du main doit être identique.

Corrigé si vous êtes coincé (sélectionner, copier, puis coller dans votre code) :



Finalement, le but de ce 1er exercice est d'*obtenir un sous-programme qui assemble et renvoie à l'appelant une chaîne « multi-lignes » qui correspond à une séquence de lignes saisies par l'utilisateur, la fin de la saisie étant indiqué par une ligne vide.* Exemple, avec ce que l'utilisateur entre à gauche et la chaîne saisie au milieu : c'est à dire exactement ce que l'utilisateur a entré sauf la dernière ligne vide qui sert juste à indiquer qu'on a terminé de saisir. À l'affichage on peut ajouter des caractères encadrant la chaîne reçue, par exemple " juste avant et juste après la chaîne pour vérifier qu'on n'a pas un retour ligne en trop. À droite corrigé complet.

Saisie de l'utilisateur	Résultat reçu et affiché par l'appelant encadré par des "	Corrigé
patati patata⏎ yodli yodla⏎ ⏎	"patati patata yodli yodla "	

Application « à la carte »

L'objectif qu'on se donne maintenant est une simulation de communication wifi entre un terminal de saisie embarqué pour des commandes de boisson en terrasse d'une brasserie parisienne et un terminal au bar, lu par fred le barman. Le terminal de saisie des commandes correspondra à la fonction **foo** qu'on vient d'étudier, le terminal de réception sera un sous-programme qui s'appellera logiquement **bar**, et le sous-programme barman sera **fred**. L'échange entre foo et bar se fait par un paquet internet, il faut donc qu'il se fasse sous forme de flot de caractères : foo envoie une chaîne, bar reçoit une chaîne. La commande sera décomposée par bar en une succession d'appels à fred, un appel par ligne, et pour chaque ligne fred trouvera le nombre et le nom de boisson, il affichera ce nombre de fois le nom de la boisson entre crochets (dans un verre ou dans une tasse quoi !). On ne traitera pas le problème des pluriels : le serveur en terrasse est prié de ne pas mettre s à la fin des noms de boissons.

Saisie par foo en terrasse	Chaîne envoyée par foo reçue par bar	Envoyés successivement à fred par bar	Servi (affiché !) par fred à chaque ligne de commande

`2 chocolat⏎`
`3 diabolo menthe⏎`
`1 planteur⏎`
	`"2 chocolat`
	`3 diabolo menthe`
	`1 planteur`
	`"`
		`"2 chocolat"` →
		`"3 diabolo menthe"` →
		`"1 planteur"` →
			`[chocolat]`
			`[chocolat]`
			`[diabolo menthe]`
			`[diabolo menthe]`
			`[diabolo menthe]`
			`[planteur]`

Créez un nouveau projet C++, appelez le ***`a_la_carte`*** *par exemple...*

2. Parsing de chaîne, `std::istringstream`

Commençons par **fred** le barman. Le but est de décomposer une chaîne reçue en morceaux, pour y retrouver des valeurs numériques ou textuelles. On va **traiter la chaîne entrante comme std::cin**. Ce n'est pas possible directement depuis une std::string mais c'est possible avec un objet std::istringstream, lequel peut être initialisé facilement à partir d'une string. Si vous pensez avoir compris le principe passez directement à la version 2, sinon faites la version 1 d'abord.

Version 1 : mise en place avec std::cin. On va avoir un sous programme sans paramètre entrant et sans return, qui lit à la console (avec std::cin) un nombre entier puis une chaine. Il affiche nombre fois cette chaîne entre crochets, séparé par des retours ligne. Par exemple quand on appelle ce sous programme et qu'on saisi à la console « 2 » puis « chocolat » il affiche directement

```
[chocolat]
[chocolat]
```

Pas de corrigé … c'est en principe assez simple. À ce stade on ne demande pas encore de gérer les noms de boissons en plusieurs mots (diabolo menthe), on verra plus tard.

Version 2 : maintenant le sous-programme fred reçoit un paramètre `string ligne` qui contient, par exemple, « 2 chocolat », et il doit d'abord déclarer une `istringstream iss` avec cette chaîne en paramètre du constructeur puis tout se passe ensuite comme pour une saisie directe avec std::cin sauf qu'on utilise iss à la place. C'est tout ! À ce stade on ne demande pas encore de gérer les noms de boissons en plusieurs mots. Pour tester on fait l'appel suivant depuis le main : `fred("2 chocolat");` // Corrigé page suivante.

Version 3 : maintenant le sous-programme fred va gérer les noms de boissons en plusieurs mots (vous pouvez tester que la version 2 précédente ne marche pas bien dans ce cas). Il suffit de **remplacer** la saisie du nom de la boisson qui se faisait avec iss >> nomBoisson; par un getline :

iss.ignore(); // Saute l'espace qui séparait le nombre du nom de la boisson
std::getline(iss, nomBoisson); // Lire le **reste de la ligne** comme étant le nom de la boisson
Tester avec l'appel depuis le main `fred("3 diabolo menthe");`

3. Lecture ligne par ligne d'un flot entrant

On va finaliser l'application avec le sous-programme **bar**. Il reçoit une chaîne « multi-ligne » et appel fred avec chaque ligne successivement. La lecture ligne par ligne d'un flot entrant se fait en enchaînant des getline. Quand le flot se termine parce qu'on est à la fin d'un istringstream ou la fin d'un ifstream la fonction getline échoue et renvoie **false**[2]. La lecture ligne par ligne d'un fichier ou d'un istringstream prend donc en général la forme suivante :

```
std::string ligne;
while ( std::getline(flotEntrant, ligne) )
{
    // faire quelque chose avec la ligne récupérée
}
```

Utiliser ce schéma de lecture ligne par ligne pour écrire le sous-programme bar. Le tester avec:

```
bar("2 chocolat\n3 diabolo menthe\n1 planteur\n"); // depuis main
```

Finalement le main de l'application qui permet de saisir interactivement une commande et de la passer au bar et de se faire servir (attention aux abus...) :

```
int main()
{
    /// Récupération d'une commande sur le terminal du serveur
    std::string commande = foo();

    /// La commande est sous forme de chaîne de caractère
    std::cout << "Le paquet qui passe par le wifi :\n"
              << "<<\n" << commande << ">>\n\n";

    /// La commande est envoyée au bar pour être préparée...
    bar(commande);

    return 0;
}
```

2 Techniquement getline retourne une référence au flot entrant, mais dans un contexte de test cette référence est implicitement convertie en pointeur qui est lui même converti en bool. *No comment.*

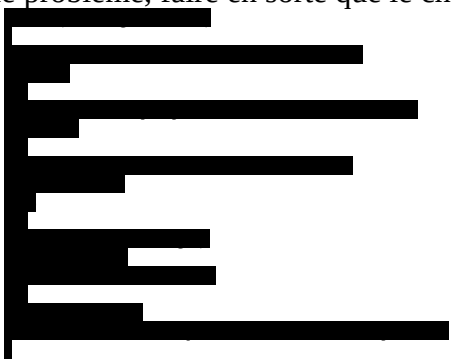
4. *Sauver fichier,* `std::ofstream`

A partir du code déjà réalisé on n'est pas loin de pouvoir faire transiter la commande par un fichier plutôt que par une chaîne. Supposons que la programmation de l'envoi direct d'un paquet sur le wifi s'avère trop compliquée et qu'on préfère passer par un lecteur réseau : la commande en terrasse passera bien par le wifi mais en tant que fichier. On veut alors maintenant que **foo** en terrasse écrive la commande dans un **fichier** et que **bar** lise ce fichier. Pour ne pas se mélanger avec le projet précédent, *créez un nouveau projet C++, appelez le* ***`fichier_en_terrasse`*** *par exemple... Copiez-collez l'intégralité du code précédent.*

Modifier foo pour qu'il écrive un fichier au lieu de retourner une chaîne : foo ne retourne plus rien et le nom du fichier sera reçu en paramètre entrant. Dans foo, au lieu d'un ostringstream on utilise un objet flot de sortie vers fichier **ofstream**. La déclaration (le constructeur) de ce dernier nécessite un paramètre nom de fichier : lui passer le nom reçu par foo ! Le nom d'un ostringstream est traditionnellement oss, pour un ofstring on pourra le renommer en **ofs**. Enfin comme l'ouverture d'un fichier est toujours un processus dont la réussite dépend du « monde extérieur » (le répertoire n'existe pas, le fichier est verrouillé par une autre application, le lecteur réseau est inaccessible...) il faut tester si il y a un problème. Comment réagir ? Ça dépend de ce qu'on veut faire pour remédier au problème, pour l'instant on peut se contenter d'indiquer qu'il y a un problème avec un message std::cerr << … et faire un return anticipé (ou mettre le reste du sous-programme dans un else). On verra les exceptions à l'exo 7.

Tester avec ce code appelant dans le main : `foo("commande.txt");` et vérifier que 2 ou 3 lignes de commandes saisies en terrasse arrivent bien dans le fichier « commande.txt » du répertoire d'exécution (pour macOS avec Xcode c'est comme pour trouver output.svg...)

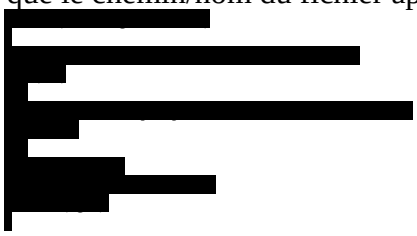
Tester avec ce code appelant dans le main : `foo("inexistant/commande.txt");` C'est à dire qu'on demande de créer et d'écrire un fichier dans un sous-répertoire qui n'existe pas. Vérifier que la gestion d'erreur est correcte : la saisie n'a pas lieu et on a bien le message d'erreur prévu en cas de problème, faire en sorte que le chemin/nom du fichier apparaisse dans le message.



5. *Charger fichier,* `std::ifstream`

Modifier bar pour qu'il lise un fichier au lieu de parser une chaîne : au lieu d'une chaîne contenant la commande, le nom du fichier sera reçu en paramètre entrant. Dans bar, au lieu d'un istringstream on utilise un objet flot d'entrée depuis fichier **ifstream**. La déclaration (le constructeur) de ce dernier nécessite un paramètre nom de fichier : lui passer le nom reçu par bar ! Le nom d'un istringstream est traditionnellement iss, pour un ifstring on pourra le renommer en **ifs**. Enfin comme l'ouverture d'un fichier est toujours un processus dont la réussite dépend du « monde extérieur » (le fichier à lire n'existe pas ou est inaccessible...) il faut tester si il y a un problème. Comment réagir ? Ça dépend de ce qu'on veut faire pour remédier au problème, pour l'instant on peut se contenter d'indiquer qu'il y a un problème avec un message std::cerr << … et faire un return anticipé (ou mettre le reste du sous-programme dans un else). On verra les exceptions à l'exo 7.

Neutraliser dans le main l'appel à foo pour tester séparément bar avec ce code appelant : `bar("commande.txt");` et vérifier que la commande saisie en terrasse lors de l'exécution précédente est bien lue depuis le fichier « commande.txt ». Vérifier que la gestion d'erreur est correcte : si on remplace "commande.txt" par un nom de fichier qui n'existe pas alors fred n'est pas appelé (pas de boisson servie) et on a bien le message d'erreur prévu en cas de problème, faire en sorte que le chemin/nom du fichier apparaisse dans le message.



6. Facultatif : gestion avancée des anomalies

L'énoncé de cet exo est une indication de ce qu'on pourrait vouloir faire pour aller au delà en matière de gestion des anomalies pour l'application « fichier_en_terrasse ». À moins d'être très productif et gravement motivé il est recommandé d'étudier la gestion des exceptions à l'exo 7, plus simple et plus détaillé.

Il faudrait pour être plus complet dans la couverture des anomalies remplacer la gestion des problèmes d'ouvertures des fichiers par le lancement d'exceptions (runtime_error). D'autre part il n'est pas évident que le format reçu soit correct : dans fred tester la bonne lecture des éléments dans la ligne en vérifiant .fail() sur le flot istringstream, lancer une exception si nécessaire. Il est préférable de traiter les problèmes en amont, et il est préférable de ne pas lancer une exception quand on peut l'éviter. Lors de la saisie de la commande (dans foo) il serait souhaitable de faire une vérification du bon format de chaque ligne saisie (à mesure qu'elles arrivent). Une fonction commune utilisée par foo et par fred pour valider/parser une ligne serait utile. Mais quand un problème de parsing de ligne de commande arrive dans foo, plutôt que de lancer une exception on peut re-demander une nouvelle saisie selon le « blindage » classique. Sauf que contrairement à un blindage classique on voudra autoriser le client qui fait la saisie à ne pas rester bloqué : il a le droit d'indiquer qu'il veut annuler sa commande et « aller dans un bar plus sympa ». Dans ce cas on s'en sort avec une exception. Enfin plutôt que de « servir » (afficher) une partie de la commande avant de se rendre compte qu'il y a un problème (fichier corrompu...), le bar devrait attendre que le plateau soit complet : ceci implique que fred ne travaille plus directement dans std::cout mais vers un ostringstream& plateau reçu en paramètre par référence de bar, lequel ne livre la commande (affiche le plateau) que si la totalité de la transaction s'est bien déroulée ! Naturellement il ne suffit pas de lancer des exceptions, encore faut-il les gérer, donc prévoir un/des blocs try/catch appropriés.

7. Exceptions : gestion des cafards

*Créez un nouveau projet C++, **`oggy_et_les_cafards`** par exemple, collez le code suivant:*

```
#include <iostream>
#include <string>

std::string preparerDinerRomantique();
std::string recupererCave(std::string quoi);

int main()
{
    std::string tableSalon;
    tableSalon = preparerDinerRomantique();
    std::cout << tableSalon << std::endl;

    return 0;
}

std::string preparerDinerRomantique()
{
    std::string table;
    table += recupererCave("vin") + "\n";
    table += recupererCave("decoration") + "\n";
    table += recupererCave("legumes") + "\n";
    return table;
}
```

```
std::string recupererCave(std::string quoi)
{
    if (quoi=="vin") return "Bouteille de Bordeaux";
    if (quoi=="legumes") return "Poireaux";
    return "Cafards";
}
```

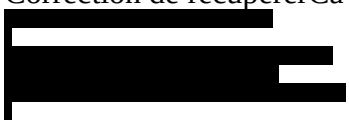
Remettre en forme (Code::Blocks menu Plugins => Source code formatter (AStyle) Tester...

Constater que notre dîner romantique aux chandelles est ruiné ! Bien sûr on ne veut **pas** juste ajouter `if (quoi=="decoration") return "chandelles";` ça résoudrait cette utilisation particulière mais pas une autre. Le problème général c'est de demander une chose qui n'a pas de correspondance : dans ce cas la valeur de retour spéciale "Cafards" n'est qu'un symptôme. On ne résout pas un problème en effaçant un symptôme : *essayer de neutraliser la ligne* avec le `return "Cafards";` et tester : ça donne quoi ? Est-ce plus satisfaisant ?

On pourrait traiter le problème en détectant la valeur spécial Cafards dans le code de plus haut niveau et en tirer les conséquences (annuler proprement le dîner) mais nous sommes là pour apprendre à utiliser le mécanisme idéal pour ce genre de situation : **les exceptions** !

A la place de `return "Cafards";` lancer une exception de type logic_error avec un message associé expliquant que la chose recherchée n'est pas dans la cave. Ne pas oublier le #include <stdexcept>.

Correction de recupererCave :

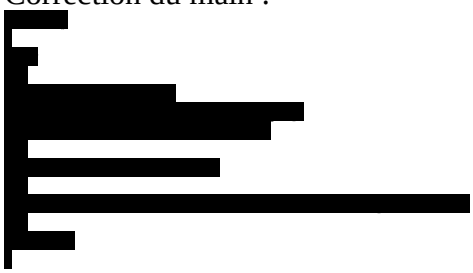


Histoire de voir ce que ça fait : voyez ce qui se passe quand une exception est lancée mais que l'appel ne vient pas d'un bloc try/catch. Et en mode debug ? (Dans Code::Blocks le triangle rouge ou menu Debug => Start / Continue. Croix rouge ou menu Debug => Stop debugger pour arrêter proprement. Le debugger peut s'arrêter ou pas sur les exceptions : en cours de debug voir menu Debug => Information => Catch throw)

Maintenant faire comme il convient avec les exceptions : *mettre le code appelant dans un bloc try/catch avec une interception de l'exception et un affichage du message remonté.* On doit sortir proprement de la situation, avec un code d'erreur nul (c'est à dire qu'on pourrait continuer le programme si on voulait) :

```
On annule le diner, raison : decoration pas dans la cave

Process returned 0 (0x0)   execution time : 0.016 s
Press any key to continue.
```

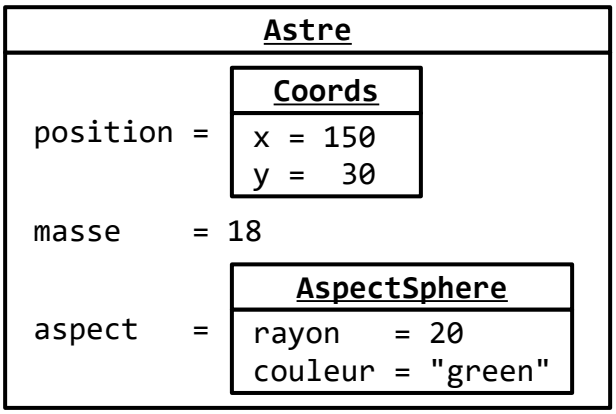
Correction du main :



8. Facultatif : sérialisation / dé-sérialisation de composites

Cet exo est long, c'est inévitable pour écrire du code de sérialisation/dé-sérialisation : c'est aussi pourquoi on préfère utiliser des bibliothèques ou frameworks pour traduire nos objets en fichiers, mais ceci dépasserait ce module POO/C++ sur un seul semestre! D'autant que certains problèmes de sauvegarde de situations complexes (entités avec double navigation) nécessitent des concepts de théorie des graphes (2ème semestre). Si toutefois vous êtes curieux de savoir comment faire « en manuel », voici quelques indications pour un simple modèle composite par valeur.

En partant et en s'inspirant du code présenté au slide 66 du cours 10 on peut écrire le code de sérialisation/dé-sérialisation pour les objets d'une classe AspectSphere. En s'appuyant sur ces méthodes des classes composantes on va pouvoir écrire les méthodes équivalentes dans une classe composite Astre pour arriver au modèle suivant (utilisé au TD/TP 5) :



Tester au fur et à mesure. Grâce aux types polymorphes istream et ostream on peut indifféremment utiliser ces méthodes avec std::cin pour tester en manuel, ou un istringstream pour ne pas avoir à retaper à chaque fois mais sans avoir à accéder à un vrai fichier ce qui est commode, et enfin sans modification du code des classes tester sur fichier. Voir slide 67.

Modifier les méthodes de sérialisation si nécessaire (remplacer des std::cout par de simples espaces) pour que les données sérialisées d'un seul Astre soient sur une même ligne (pas de \n), ceci nous donnera un fichier nettement plus lisible. Mettre en place une classe composite de plus haut niveau : Systeme. Un objet de la classe Systeme encapsulera un vecteur ou une liste d'objets Astre (gérés par valeur ou par adresses obtenues dynamiquement avec new). Coder les méthodes de sérialisation de la classe système. En ignorant la gestion d'erreurs on peut arriver à la structure suivante pour le main qui permet de retrouver le même système de session en session :

```
int main()
{
    std::ifstream ifs{"systeme.txt"};
    Systeme principal{ifs}
    ifs.close();

    int choixMenu;
    do
    {
        /// faire choix menu, switch, ajouter/editer des Astres...
    }

    std::ofstream ofs{"systeme.txt"};
    principal.serialize{ofs};
    ofs.close();

    return 0;
}
```

Dans le fichier on trouvera une ligne par objet de type Astre. Avec ce genre de liste d'objets, c'est souvent une approche efficace d'avoir prévu (à la sauvegarde) dans le fichier le nombre des objets avant les données de ces objets.

Cette structure du main reste rudimentaire. Il convient d'ajouter une gestion d'erreurs solide avec les exceptions, en particulier au chargement, et de ne pas écraser systeme.txt avec les données d'un système principal « par défaut » c'est à dire vide ou presque (le remède serait pire que le mal!)

Il existe de nombreuses variantes. Dans les classes on peut avoir un constructeur par défaut et une méthode unserialize plutôt que de faire la dé-sérialisation directement dans le constructeur (ce qui peut poser des problèmes d'ordre, de logique de contrôle trop difficile à faire dans une liste d'initialisation...). On peut surcharger les opérateurs << et >> pour donner à nos types un usage similaire aux types de la bibliothèque standard : on pourra alors sauver ou afficher un objet avec un simple ofs << astre[i] ou std::cout << astre[i] (voir cours 4 surcharge d'opérateurs, avec des structs sur l'exemple, avec une classe il convient de déclarer friend l'opérateur dans la classe)