



C++ - Módulo 04

Polimorfismo de subtipado, clases abstractas,
interfaces

Resumen: Este documento contiene la evaluación del módulo 04 de los módulos C++ de 42.

Índice general

I.	Reglas Generales	2
II.	Ejercicio 00: Polimorfismo, o “Cuando el hechicero considera que estás más mono transformado en oveja”	4
III.	Ejercicio 01: No quiero quemar el mundo	8
IV.	Ejercicio 02: This code is unclean. Purify it!	14
V.	Ejercicio 03: Bocal Fantasy	17
VI.	Ejercicio 04: AFK Mining	21

Capítulo I


Reglas Generales

- La declaración de una función en un header (excepto para los templates) o la inclusión de un header no protegido conllevará un 0 en el ejercicio.
- Salvo que se indique lo contrario, cualquier salida se mostrará en stdout y terminará con un newline.
- Los nombres de ficheros impuestos deben seguirse escrupulosamente, así como los nombres de clase, de función y de método.
- Recordatorio : ahora está codificando en C++, no en C. Por eso :
 - Las funciones siguientes están **PROHIBIDAS**, y su uso conllevará un 0: `*alloc`, `*printf` et `free`
 - Puede utilizar prácticamente toda la librería estándar. NO OBSTANTE, sería más inteligente intentar usar la versión para C++ que a lo que ya está acostumbrado en C, para no basarse en lo que ya ha asimilado. Y no está autorizado a utilizar la STL hasta que le llegue el momento de trabajar con ella (módulo 08). Eso significa que hasta entonces no se puede utilizar Vector/List/Map/etc... ni nada similar que requiera un include `<algorithm>`.
- El uso de una función o de una mecánica explícitamente prohibida será sancionado con un 0
- Tenga también en cuenta que, a menos que se autorice de manera expresa, las palabras clave `using namespace` y `friend` están prohibidas. Su uso será castigado con un 0.
- Los ficheros asociados a una clase se llamarán siempre `ClassName.cpp` y `ClassName.hpp`, a menos que se indique otra cosa.
- Tiene que leer los ejemplos en detalle. Pueden contener prerrequisitos no indicados en las instrucciones.
- No está permitido el uso de librerías externas, de las que forman parte C++11, Boost, ni ninguna de las herramientas que ese amigo suyo que es un figura le ha recomendado.
- Probablemente tenga que entregar muchos ficheros de clase, lo que le va a parecer repetitivo hasta que aprenda a hacer un script con su editor de código favorito.

- Lea cada ejercicio en su totalidad antes de empezar a resolverlo.
- El compilador es `clang++`
- Se compilará su código con los flags `-Wall -Wextra -Werror`
- Se debe poder incluir cada `include` con independencia de los demás `include`. Por lo tanto, un `include` debe incluir todas sus dependencias.
- No está obligado a respetar ninguna norma en C++. Puede utilizar el estilo que prefiera. Ahora bien, un código ilegible es un código que no se puede calificar.
- Importante: no va a ser calificado por un programa (a menos que el enunciado especifique lo contrario). Eso quiere decir que dispone cierto grado de libertad en el método que elija para resolver sus ejercicios.
- Tenga cuidado con las obligaciones, y no sea zángano; podría dejar escapar mucho de lo que los ejercicios le ofrecen.
- Si tiene ficheros adicionales, no es un problema. Puede decidir separar el código de lo que se le pide en varios ficheros, siempre que no haya moulinette.
- Aun cuando un enunciado sea corto, merece la pena dedicarle algo de tiempo, para asegurarse de que comprende bien lo que se espera de usted, y de que lo ha hecho de la mejor manera posible.

Capítulo II

Ejercicio 00: Polimorfismo, o “Cuando el hechicero considera que estás más mono transformado en oveja”

	Ejercicio : 00
Polimorfismo, o “Cuando el hechicero considera que estás más mono transformado en oveja”	
Directorio de entrega : <i>ex00/</i>	
Ficheros a entregar : <i>Sorcerer.hpp</i> , <i>Sorcerer.cpp</i> , <i>Victim.hpp</i> , <i>Victim.cpp</i> , <i>Peon.hpp</i> , <i>Peon.cpp</i> , <i>main.cpp</i>	
Funciones prohibidas : Ninguna	

El polimorfismo es una tradición ancestral que viene de la época de los magos, hechiceros y otros charlatanes. Le podríamos decir que fuimos los primeros en inventarlo, ¡pero le estaríamos mintiendo!

Centrémonos en nuestro amigo Ro/b/ert, el Magnífico, de profesión hechicero.

Robert tiene una afición interesante: transformar todo lo que pilla: ovejas, ponis, nutrias y muchas más cosas improbables (¿Nunca ha visto un grifo...?).

Comencemos creando la clase *Sorcerer*, con un nombre y un título. Tiene un constructor que recibe el nombre y el título como parámetros (en ese orden).

No se puede instanciar sin parámetros (¡No tendría ningún sentido! Imagínese un hechicero sin nombre ni título... Pobrecillo, no podría fardar frente a las chicas de la taberna...). Pero tendrá que usar siempre la forma de Coplien. Sí, aquí también hay truco.

Al nacer, el hechicero muestra:

```
NOM, TITRE, is born!
```

(Por supuesto, tendrá que remplazar NOM y TITRE con el nombre del hechicero y su título, respectivamente).

Al morir, muestra:

```
NOM, TITRE, is dead. Consequences will never be the same!
```

Un hechicero se debe poder presentar como es debido:

```
I am NOM, TITRE, and i like ponies!
```

Se puede presentar sobre cualquier output, con un overload del operador <<. (Recordatorio: está prohibido utilizar friend. Añada los getters que hagan falta).

Ahora, nuestro hechicero necesita víctimas, para divertirse por la mañana, entre las garras de osos y el zumo de trol.

Por lo tanto, cree la clase `Victim`. Parecido al hechicero, tendrá un nombre y un constructor que recibirá su nombre como parámetro.

Cuando nazca la víctima, muestre:

```
A random victim called NOM just appeared!
```

Cuando muera, muestre:

```
The victim NOM died for no apparent reasons!
```

La víctima también se puede presentar, como el Hechicero, y decir:

```
I'm NOM and I like otters!
```

El `Sorcerer` puede “polymorphed()” a la `Victim`. Añada un método `void getPolymorphed() const` que diga:

```
NOM was just polymorphed in a cute little sheep!
```

Añada también la función miembro `void polymorph(Victim const &) const` al `Sorcerer` para que pueda crear polimorfos de la gente.

Ahora, al **Sorcerer** le gustaría crear polimorfos de otras cosas, para variar, y no solo de **Victim** genéricas.

¡Ningún problema! ¡Vamos a crear otras!

Cree una clase **Peon**.



Un **Peon** es una **Victim**. Por lo tanto...

Cuando nazca tendrá que decir “Zog zog.” y cuando muera “Bleuark...” (Mire el ejemplo, es más complicado de lo que parece).

El polimorfo de un **Peon** tiene que crearse de la siguiente forma:

```
NOM was just polymorphed into a pink pony!
```

(Un poNimorfo, JAJAJAJA... jajaja... ja)

El siguiente código tiene que compilar y mostrar el output correcto:

```
int main()
{
    Sorcerer robert("Robert", "the Magnificent");

    Victim jim("Jimmy");
    Peon joe("Joe");

    std::cout << robert << jim << joe;

    robert.polymorph(jim);
    robert.polymorph(joe);

    return 0;
}
```

Output :


```
$> clang++ -W -Wall -Werror *.cpp
$> ./a.out | cat -e
Robert, the Magnificent, is born!$
A random victim called Jimmy just appeared!$
A random victim called Joe just appeared!$
Zog zog.$
I am Robert, the Magnificent, and i like ponies!$
Je suis Jimmy and I like otters!$
Je suis Joe and I like otters!$
Jimmy was just polymorphed in a cute little sheep!$
Joe was just polymorphed in a cute little sheep!$
Bleuark...$
Victim Joe died for no apparent reasons!$
Victim Jimmy died for no apparent reasons!$
Robert, the Magnificent, is dead. Consequences will never be the same!$
$>
```

Si realmente es concienzudo, podría hacer más pruebas: añadir clases derivadas, etc. (¡Hágalo!)

No olvide entregar su propio main, dado que lo que no haya probado no tendrá nota.

Capítulo III

Ejercicio 01: No quiero quemar el mundo

	Ejercicio : 01
No quiero quemar el mundo	
Directorio de entrega : <i>ex01/</i>	
Ficheros a entregar : AWeapon.[hpp,cpp], PlasmaRifle.[hpp,cpp], PowerFist.[hpp,cpp], Enemy.[hpp,cpp], SuperMutant.[hpp,cpp], RadScorpion.[hpp,cpp], Character.[hpp,cpp], main.cpp	
Funciones prohibidas : Ninguna	

En los Wasteland podrá encontrar de todo. Trozos de metal, productos químicos extraños, cruces entre cowboys y punkis vagabundos, pero también un cargamento de armas improbables (¡pero graciosas!). Ya era hora, tenía ganas de partir caras hoy.

Para sobrevivir en todo este caos va a tener que programar armas. Complete e implemente la siguiente clase (no se olvide de la forma de Coplien...):

```
class AWeapon
{
    private:
        [...]

    public:
        AWeapon(std::string const & name, int apcost, int damage);
        [...] ~AWeapon();
        std::string [...] getName() const;
        int getAPCost() const;
        int getDamage() const;
        [...] void attack() const = 0;
};
```

Info:

- Las armas tienen un nombre, un valor de daños y un coste en AP (Action Points) para disparar.
- Las armas tienen ruidos y efectos visuales asociados cuando se las utiliza con `attack()`. Todo esto va gestionado en las clases heredadas.

A continuación, implemente las clases específicas `PlasmaRifle` y `PowerFist`. He aquí sus características:

- `PlasmaRifle` :
 - Name: "Plasma Rifle"
 - Damage: 21
 - AP cost: 5
 - Output of `attack()`: `"* piouuu piouuu piouuu *"`
- `PowerFist`:
 - Name: "Power Fist"
 - Damage: 50
 - AP cost: 8
 - Output of `attack()`: `"* pschhh... SBAM! *"`

¡Ya está! Ahora que ya tenemos armas para pelear, vamos a añadir enemigos que podamos machacar (aplastar, reventar, eliminar, clavar a las puertas, desintegrar, etc.).

Cree una clase **Enemy** con el modelo siguiente (Por supuesto, tiene que crear una clase de Coplien):

```
class Enemy
{
    private:
        [...]

    public:
        Enemy(int hp, std::string const & type);
        [...] ~Enemy();
        std::string [...] getType() const;
        int getHP() const;

        virtual void takeDamage(int);
};
```

Requisitos:

- Los enemigos disponen de una cantidad de puntos de vida y tienen un tipo.
- Los enemigos pueden recibir daños (que reducen sus HP). Si los daños son <0 , no haga nada.

Va a implementar algunos enemigos específicos. Para que podamos divertirnos.

En primer lugar, la clase **SuperMutant**. Gordo, feo, malo y con el coeficiente intelectual de una maceta. Un poco como un elefante en un pasillo. Si lo falla es que lo hace adrede. En definitiva, un buen punching ball para entrenarse.

He aquí sus características:

- 170
- Tipo: "Super Mutant"
- Cuando nazca, muestre: "Gaaah. Break everything !"
- Cuando muera, muestre: ".Aaargh ..."
- Haga un overload de **takeDamage** para retirar de forma permanente 3 puntos a los daños sufridos. (¡Son así de fuertes!)

Después, cree una clase **RadScorpion**. No es que sea una bestia TAN salvaje, pero aún así: Un escorpión gigante tiene su encanto, ¿verdad?

- Características:
 - 80
 - Tipo: "RadScorpion"
 - Cuando nazca, muestre: "* click click click *"
 - Cuando muera, muestre: "* SPROTCH *"

Ya tenemos armas, enemigos para probarlas, ¡solo nos falta crear al héroe!

Entonces va a crear la clase **Character** con el modelo siguiente (ya conoce la mecánica):

```
class Character
{
    private:
        [...]

    public:
        Character(std::string const & name);
        [...]
        ~Character();
        void recoverAP();
        void equip(AWeapon*);
        void attack(Enemy*);
        std::string [...] getName() const;
};
```

- Tiene un nombre, una cantidad de puntos de vida y un puntero a **AWeapon** que representa el arma con la que está equipado en ese momento.

- Cuando se crea, el héroe dispone de 40 AP y los va perdiendo cada vez que utiliza el arma. Recupera 10 AP cada vez que se llama a **recoverAP()**, con un máximo de 40 AP. Si no le quedan AP, no puede atacar.
- Muestra "NOM ataque ENEMY_TYPE with a WEAPON_NAME" cada vez que se llame a **attack()**, seguido de una llamada a **attack()** del arma actual. Si no va equipado con ningún arma, **attack()** no hace nada. Le quita HP al enemigo en función de los daños que cause el arma. Si los HP del enemigo llegan a 0, tiene que destruirlo.
- **equip()** debe almacenar únicamente un puntero al arma. No hay copia.

También tendrá que implementar un overload del operador << para mostrar las características de su personaje. Añada los getters que hagan falta.

El overload mostrará:

```
NAME has AP_NUMBER AP and carries a WEAPON_NAME
```

Si tiene un arma. Si no muestre:

```
NOM has AP_NUMBER AP and is unarmed
```

He aquí una pequeña función main (básica) para hacer pruebas. La suya tendrá que ser mejor:

```
int main()
{
    Character* moi = new Character("moi");

    std::cout << *moi;

    Enemy* b = new RadScorpion();

    AWeapon* pr = new PlasmaRifle();
    AWeapon* pf = new PowerFist();

    moi->equip(pr);
    std::cout << *moi;
    moi->equip(pf);

    moi->attack(b);
    std::cout << *moi;
    moi->equip(pr);
    std::cout << *moi;
    moi->attack(b);
    std::cout << *moi;
    moi->attack(b);
    std::cout << *moi;

    return 0;
}
```


Output :

```
$> clang++ -W -Wall -Werror *.cpp
$> ./a.out | cat -e
moi has 40 AP and is unarmed$
* click click click *$
moi has 40 AP and carries a Plasma Rifle$
moi has ttaque RadScorpion with a Power Fist$
* pschhh... SBAM! *$
moi has 32 AP and carries a Power Fist$
moi has 32 AP and carries a Plasma Rifle$
moi has ttaque RadScorpion with a Plasma Rifle$
* piouuu piouuu piouuu *$
moi has 27 AP and carries a Plasma Rifle$
moi has ttaque RadScorpion with a Plasma Rifle$
* piouuu piouuu piouuu *$
* SPROTCH *$
moi has 22 AP and carries a Plasma Rifle$
```

Como de costumbre, entregue un `main` que incluya las pruebas.

Capítulo IV

Ejercicio 02: This code is unclean. Purify it!

	Ejercicio : 02
This code is unclean. Purify it!	
Directorio de entrega : <i>ex02/</i>	
Ficheros a entregar : <code>Squad.hpp</code> , <code>Squad.cpp</code> , <code>TacticalMarine.hpp</code> , <code>TacticalMarine.cpp</code> , <code>AssaultTerminator.hpp</code> , <code>AssaultTerminator.cpp</code> , <code>ISpaceMarine.hpp</code> , <code>ISquad.hpp</code> , <code>main.cpp</code>	
Funciones prohibidas : Ninguna	

Su misión consiste en crear un ejercito cuya apariencia solo pueda compararse con su violencia.

Tendrá que implementar una clase `Squad` y una clase `TacticalMarine` (para formar su escuadrón).

Empecemos con `Squad`. He aquí la interfaz que tendrá que implementar (incluya `ISquad.hpp`):

```
class ISquad
{
    public:
        virtual ~ISquad() {}
        virtual int getCount() const = 0;
        virtual ISpaceMarine* getUnit(int) const = 0;
        virtual int push(ISpaceMarine*) = 0;
};
```

La tendrá que implementar de tal forma que:

- `getCount()` devuelva la cantidad de unidades que actualmente hay en el escuadrón.
- `getUnit(N)` devuelva un puntero a la unidad N (que empiece por 0)

- `push(XXX)` añade la unidad XXX al final de la **Squad** (No tiene ningún sentido añadir una unidad NULL o una unidad que ya se encuentre dentro de una Squad).

En el fondo, la clase **Squad** que le pedimos no es más que un contenedor para sus Marines, que utilizaremos para estructurar correctamente el ejército.

La asignación mediante copia de una **Squad** debe implicar un **deep copy**. En el caso de que ya haya alguna unidad en el escuadrón, habrá que destruirla antes de reemplazarla. Se crearán las unidades con `new()`.

Cuando se destruye una **Squad**, se destruyen también las unidades que se encuentran en el escuadrón, de forma ordenada.

He aquí la interfaz que hay que respetar para implementar los **TacticalMarine**. `ISpaceMarine.hpp` :

```
class ISpaceMarine
{
    public:
        virtual ~ISpaceMarine() {}
        virtual ISpaceMarine* clone() const = 0;
        virtual void battleCry() const = 0;
        virtual void rangedAttack() const = 0;
        virtual void meleeAttack() const = 0;
};
```

Requisitos:

- `clone()` devuelve una copia del objeto actual.
- Al nacer, muestra: "Tactical Marine ready for action!"
- `battleCry()` muestra "For the Holy PLOT!"
- `rangedAttack()` muestra "* attacks with a bolter *"
- `meleeAttack()` muestra "* attacks with a chainsword *"
- Al morir, muestra: ".^aargh ..."

Del mismo modo, implemente un **AssaultTerminator** que muestre lo siguiente:

- Nacimiento: "* teleports from space *"
- `battleCry()`: "This code is unclean. Purify it!"
- `rangedAttack`: "* does nothing *"
- `meleeAttack`: "* attaque with chainfists *"
- Muerte: "I'll be back..."

He aquí un poco de código para realizar pruebas. Como siempre, el suyo tendrá que ser más riguroso.

```
int main()
{
    ISpaceMarine* bob = new TacticalMarine;
    ISpaceMarine* jim = new AssaultTerminator;

    ISquad* vlc = new Squad;
    vlc->push(bob);
    vlc->push(jim);
    for (int i = 0; i < vlc->getCount(); ++i)
    {
        ISpaceMarine* cur = vlc->getUnit(i);
        cur->battleCry();
        cur->rangedAttack();
        cur->meleeAttack();
    }
    delete vlc;

    return 0;
}
```


Output :

```
$> clang++ -W -Wall -Werror *.cpp
$> ./a.out | cat -e
Tactical Marine ready for action!$
* teleports from space *$
For the Holy PLOT!$
* attacks with a bolter *$
* attaque with chainsword *$
This code is unclean. Purify it!$
* does nothing *$
* attaque with chainfists *$
Aaargh ...$
I'll be back ...$
```

Si quiere una buena nota, dedíquele tiempo al `main`.

Capítulo V

Ejercicio 03: Bocal Fantasy

	Ejercicio : 03
Bocal Fantasy	
Directorio de entrega : <i>ex03/</i>	
Ficheros a entregar : <i>AMateria.hpp, AMateria.cpp, Ice.hpp, Ice.cpp, Cure.hpp, Cure.cpp, Character.hpp, Character.cpp, MateriaSource.hpp, MateriaSource.cpp, ICharacter.hpp, IMateriaSource.hpp, main.cpp</i>	
Funciones prohibidas : Ninguna	

Completa la definición de la clase `AMateria`, y implemente las funciones miembros necesarias.

```
class AMateria
{
private:
    [...]
    unsigned int _xp;

public:
    AMateria(std::string const & type);
    [...]
    [...] ~AMateria();

    std::string const & getType() const; //Returns the materia type
    unsigned int getXP() const; //Returns the Materia's XP

    virtual AMateria* clone() const = 0;
    virtual void use(ICharacter& target);
```

El sistema de experiencia de una Materia funciona de la siguiente forma:

- Los XP totales de una Materia empiezan con 0 y aumentan en 10 puntos cada vez que se llama a `use()`. Encuentre un modo inteligente de gestionarlo.

Cree las clases específicas **Ice** y **Cure**. Su tipo será el nombre de la clase en minúsculas. Por supuesto, su método `clone()` tendrá que devolver una instancia nueva del verdadero tipo de **Materia**.

El método `use(ICharacter&)` mostrará:

- Ice: `"* shoots an ice bolt at NOM *"`
- Cure: `"* heals NOM's wounds *"`

(Obviamente, NOM será remplazado por el **Character** que se reciba como parámetro).



No tiene sentido copiar el tipo cuando asigna una **Materia** a otra.

Cree la clase **Character** que implemente la siguiente interfaz:

```
class ICharacter
{
    public:
        virtual ~ICharacter() {}
        virtual std::string const & getName() const = 0;
        virtual void equip(AMateria* m) = 0;
        virtual void unequip(int idx) = 0;
        virtual void use(int idx, ICharacter& target) = 0;
};
```

El **Character** dispone de un inventario que contiene hasta 4 **Materia**. Comienza vacío. Va adquiriendo las **Materia** de forma ordenada, de 0 a 3.

Si el inventario está lleno y se intenta equipar al personaje con una **Materia** o utilizar/desequipar una **Materia** que no exista, no haga nada. `unequip()` NO debe suprimir ninguna **Materia**.

`use(int, ICharacter&)` tendrá que utilizar la **Materia** en el slot del cual se haya indicado el índice, sobre el objetivo indicado. Pase el objetivo como parámetro al método `AMateria::use`.



Claro está, tiene que aceptar cualquier tipo de **AMateria** en su inventario.

El **Character** debe tener un constructor que reciba su nombre como parámetro. Por supuesto, la asignación mediante copia tiene que ser una **deepcopy**. Se tendrá que suprimir la antigua **Materia** del **Character**. Lo mismo cuando se destruya un **Character**.

Ahora que su personaje puede ir equipado con Materia y utilizarla, la cosa pinta mejor.

Dicho esto, no tenemos ninguna gana de crear esta Materia a mano. Cree una clase `MateriaSource` que implemente la siguiente interfaz:

```
class IMateriaSource
{
    public:
        virtual ~IMateriaSource() {}
        virtual void learnMateria(AMateria*) = 0;
        virtual AMateria* createMateria(std::string const & type) = 0;
};
```

`learnMateria` tiene que copiar la Materia pasada como parámetro y almacenarla en la memoria para poderla clonar más adelante. Igual que para `Character`, la Source no puede conocer más de 4 Materia. Además, no hace falta que sean únicas.

`createMateria(std::string const &)` devuelve una Materia nueva, que será una copia de la Materia (aprendida con anterioridad) cuyo tipo corresponde al parámetro. Si no se conoce el tipo devuelva 0.

En conclusión, Source tiene que ser capaz de aprender “templates” de Materia y volverlas a crear cuando se lo pidan. Será capaz de crear Materia sin conocer su verdadero tipo, solo a partir de un string que lo identifica.

He aquí el main inicial sobre el que tendrá que trabajar:

```
int main()
{
    IMateriaSource* src = new MateriaSource();
    src->learnMateria(new Ice());
    src->learnMateria(new Cure());

    ICharacter* moi = new Character("moi");

    AMateria* tmp;
    tmp = src->createMateria("ice");
    moi->equip(tmp);
    tmp = src->createMateria("cure");
    moi->equip(tmp);

    ICharacter* bob = new Character("bob");

    moi->use(0, *bob);
    moi->use(1, *bob);

    delete bob;
    delete moi;
    delete src;

    return 0;
}
```


Output :

```
$> clang++ -W -Wall -Werror *.cpp
$> ./a.out | cat -e
* shoots an ice bolt at bob *$
* heals bob's wounds *$
```

No se olvide de entregar su propio main.

Capítulo VI

Ejercicio 04: AFK Mining

	Ejercicio : 04
AFK Mining	
Directorio de entrega : <i>ex04/</i>	
Ficheros a entregar : <code>DeepCoreMiner.[hpp,cpp]</code> , <code>StripMiner.[hpp,cpp]</code> , <code>AsteroKreog.[hpp,cpp]</code> , <code>KoalaSteroid.[hpp,cpp]</code> , <code>MiningBarge.[hpp,cpp]</code> , <code>IAsteroid.hpp</code> , <code>IMiningLaser.hpp</code> , <code>main.cpp</code>	
Funciones prohibidas : <code>typeid()</code> u otras, lea los warnings	



Para este ejercicio, está totalmente prohibido utilizar `typeid()`.

A primera vista, podría pensar que el espacio que se encuentra más allá de Kreog es un enorme vacío. Pero no, estimado caballero. En realidad contiene una increíble cantidad de cosas aleatorias e inútiles.

Entre tías buenas del espacio puede hallar monstruos espantosos, basureros espaciales e incluso horrorosos desarrolladores de kernel. También encontrará una cantidad astronómica de asteroides repletos de minerales, a cuál más precioso. Un poco como en la fiebre del oro, pero sin el Tío Gilito.

Ahora, es explorador espacial. Acaba de llegar. Si no quiere que le tomen por un Principiante, va a necesitar herramientas. Como los picos son para los principiantes, vamos a utilizar láseres.

He aquí la interfaz que tendrá que implementar para los láseres de voladura:

```
class IMiningLaser
{
    public:
        virtual ~IMiningLaser() {}
        virtual void mine(IAsteroid*) = 0;
};
```

Implemente las dos clases siguientes: `DeepCoreMiner` y `StripMiner`. Su `mine(IAsteroid*)` dará los siguientes resultados:

- `DeepCoreMiner`

```
"* mining deep ... got RESULT ! *"
```

- `StripMiner`

```
"* strip mining ... got RESULT ! *"
```

Reemplace `RESULT` con el valor de retorno de `beMined` que llegue del asteroide objetivo. Vaya, parece que vamos a necesitar asteroides si queremos hacer voladuras. He aquí la interfaz:

```
class IAsteroid
{
    public:
        virtual ~IAsteroid() {}
        virtual std::string beMined(...) const = 0;
        [...]
        virtual std::string getName() const = 0;
};
```

Los dos asteroides que tiene que implementar son: el `Asteroid` y el `Comet`. Su método `getName()` devolverá su nombre (obvio) que será el nombre de la clase.

Utilizando el subtipado y el polimorfismo paramétrico (y su cerebro), tendrá que conseguir que una llamada a `IMiningLaser::mine` devuelva un resultado distinto en función del tipo de láser y del tipo de asteroide.

Los valores de retorno serán los siguientes:

- `StripMiner` y `Comet`: "Tartarite"

- DeepCoreMiner y Comet: "Mithril"
- StripMiner y Asteroid: "Flavium"
- DeepCoreMiner y Asteroid: "Dragonite"

Para llegar a este resultado, tendrá que completar la interfaz `IAsteroid`.



Probablemente necesite dos métodos `beMined...`



No intente deducir el valor de retorno del asteroide a través de `getName()`. TIENE que utilizar los TIPOS y el POLIMORFISMO. Está prohibido cualquier otro tipo de enfoque (`typeid`, `dynamic_cast`, `getName`, etc.).

No es tan difícil como parece. Ahora que nuestros juguetes están listos, contrúyase una nave simpática para ir a explotar rocas. Implemente la siguiente clase:

```
class MiningBarge
{
    public:
        void equip(IMiningLaser*);
        void mine(IAsteroid*) const;
};
```

- Al principio, la nave no tiene láser, pero se le pueden instalar hasta 4.
- Si ya tiene 4 láseres, `equip(IMiningLaser*)` no hace nada.
- El método `mine(IAsteroid*)` llama al `IMiningLaser::mine` de cada láser instalado, en el orden en el que han sido instalados.

Good luck.