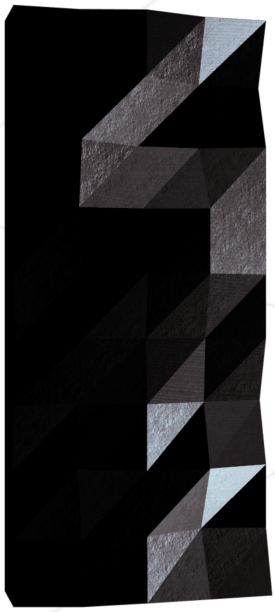

C++ - Módulo 04

Polimorfismo de subtipado, clases abstractas, interfaces

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

Índice general

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 : *ex*00/	
Ficheros a entregar : `Sorcerer.hpp, Sorcerer.cpp, Victim.hpp, Victim.cpp, Peon.hpp, Peon.cpp, main.cpp, más los archivos necesarios para sus pruebas`	
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 mintiento!

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

Robert tiene una afición interesante: transformar todo lo que tenga a la mano en, 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).

La clase 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:

```
NAME, TITLE, is born!
```

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

Al morir, muestra:

```
NAME, TITLE, is dead. Consequences will never be the same!
```

Un hechicero se debe poder presentar como es debido:

```
I am NAME, TITLE, and I like ponies!
```

Se puede presentar sobre cualquier output, con un overload del operador << a ostream (¡usted sabe como hacerlo!).
(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:

```
Some random victim called NAME just appeared!
```

Cuando muera, muestre:

```
Victim NAME died for no apparent reasons!
```

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

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

La `Victim` puede ser "polymorphed()" por el `Sorcerer`. Añada un método `void getPolymorphed() const` a la `Victim` que diga:

```
NAME has been turned into 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:

```
NAME has been turned 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
$> ./a.out | cat -e
Robert, the Magnificent, is born!$
Some random victim called Jimmy just appeared!$
Some random victim called Joe just appeared!$
Zog zog.$
I am Robert, the Magnificent, and I like ponies!$
I'm Jimmy and I like otters!$
I'm Joe and I like otters!$
Jimmy has been turned into a cute little sheep!$
Joe has been turned into a pink pony!$
Bleuark...$
Victim Joe just died for no apparent reason!$
Victim Jimmy just died for no apparent reason!$
Robert, the Magnificent, is dead. Consequences will never be the same!$
$>
```

Si realmente es concienzudo, podría hacer más pruebas: añadir clases derivadas, etc. (No, en realidad no es una sugerencia, debería hacerlo.)

Por supuesto, como siempre, entregue su propio `main`, porque todo lo que no haya probado no se calificará.

Capítulo III

Ejercicio 01: No quiero quemar el mundo

	Ejercicio : 01
No quiero quemar el mundo	
Directorio de entrega : $ex01/$	
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`, más los archivos necesarios para sus pruebas	
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
{
        protected:
                [...]

        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, una capacidad específica de hacer daño por cada impacto, y un coste en AP (Action Points) para disparar.
- Las armas tienen ruidos y efectos visuales asociados cuando se utilizan 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
{
        protected:
                [...]

        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. Un enemigo tampco debe bajar de 0 HP.

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 a propósito. En definitiva, un buen punching ball para entrenarse.

He aquí sus características:

- HP: 170
- Tipo: “Super Mutant”
- Cuando nazca, muestre: “Gaaah. Me want smash heads!”
- Cuando muera, muestre: “Aaargh ...”
- Haga un overload de `takeDamage` para recibir 3 puntos de daño menos de lo normal. (Sí, 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:
  - HP: 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 número de AP (Action points) 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()`, hasta un máximo de 40 AP. Si no le quedan AP, no puede atacar.
- Muestra "NAME attacks ENEMY_TYPE with a WEAPON_NAME" cada vez que se llame a `attack()`, seguido de una llamada al método `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 << a ostream para mostrar las características de su `Character`. Añada los getters que hagan falta.

El overload mostrará:

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

Si no tiene arma muestre:

```
NAME 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* me = new Character("me");

	std::cout << *me;

	Enemy* b = new RadScorpion();

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

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

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

	return 0;
}
```

Output :

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

Diviértase creando más armas y enemigos diferentes y entregue un `main` con sus pruebas.

Capítulo IV

Ejercicio 02: Este código está sucio. ¡PURIFÍQUELO!

	Ejercicio : 02
Este código está sucio. ¡PURIFÍQUELO!	
Directorio de entrega : $ex02/$	
Ficheros a entregar : `Squad.hpp, Squad.cpp, TacticalMarine.hpp, TacticalMarine.cpp, AssaultTerminator.hpp, AssaultTerminator.cpp, ISpaceMarine.hpp, ISquad.hpp, main.cpp`	
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 `Squad`.
- `getUnit(N)` devuelva un puntero a la unidad N (Por supuesto, empezamos en 0. Un puntero a null si el índice esta fuera de límites.)

- `push(XXX)` añade la unidad XXX al final del `Squad`. Regresa el número de unidades presentes en el `Squad` (No tiene ningún sentido añadir una unidad NULL o una unidad que ya se encuentre dentro del `Squad`).

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

Al construir una copia o asignar un `Squad`, la copia debe ser profunda. Al asignar, si había alguna unidad en el `Squad` antes, debe ser destruida antes de ser reemplazada. Se puede suponer que cada unidad se creará 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`. (Incluya `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 battle!"
- `battleCry()` muestra "For the Holy PLOT!"
- `rangedAttack()` muestra "* attacks with a bolter *"
- `meleeAttack()` muestra "* attacks with a chainsword *"
- Al morir, muestra: "Aaargh ..."

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`: "* attacks 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
$> ./a.out | cat -e
Tactical Marine ready for battle!$
* teleports from space *$
For the holy PLOT!$
* attacks with a bolter *$
* attacks with a chainsword *$
This code is unclean. PURIFY IT!$
* does nothing *$
* attacks 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 : $ex03/$	
Ficheros a entregar : `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`	
Funciones prohibidas : `Ninguna`	

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

```
class AMateria
	{
	protected:
		[...]
		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. ("ice" para Ice, etc...)

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 NAME *"
- Cure: "* heals NAME's wounds *"

(Obviamente, NAME 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 que comienza vacío. Él equipará las `Materia` en los slots 0-3, en ese orden.

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.

El método `unequip()` NO debe suprimir ninguna Materia.

El método `use(int, ICharacter&)` tendrá que utilizar la Materia en el slot del cual se haya indicado el índice. Pase el `target` 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 o copia de un `Character` tiene que ser profunda. 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 por la Source) 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. De esta manera podrá crear Materia sin conocer su “verdadero” tipo, solo a partir de un string que lo identifica. La vida es bella, ¿eh?

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* me = new Character("me");

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

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

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

    delete bob;
    delete me;
    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 : $ex04/$	
Ficheros a entregar : `DeepCoreMiner.[hpp,cpp]`, `StripMiner.[hpp,cpp]`, `AsteroKreog.[hpp,cpp]`, `KoalaSteroid.[hpp,cpp]`, `MiningBarge.[hpp,cpp]`, `IAsteroid.hpp`, `IMiningLaser.hpp`, `main.cpp`	
Funciones prohibidas : `typeid() u otras, lea los warnings`	

`Este ejercicio no puntúa, pero puede resultar interesante para su piscina. No está obligado a hacerlo.`

`Para este ejercicio, utilizar typeid() está totalmente PROHIBIDO y resultará en un -42 en este modulo. Porque sería trampa, y las trampas están mal.`

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

Entre tías buenas del espacio, monstruos espantosos, basureros espaciales e incluso horrorosos desarrolladores de kernel, 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 aquí está, recién iniciado en la exploración espacial. 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 ! *"
```

Remplace `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, esperemos), tendrá que conseguir que una llamada a `IMiningLaser::mine` devuelva un resultado distinto en función del tipo de asteroide Y del tipo de láser.

Los valores de retorno serán los siguientes:

- `StripMiner` y `Comet`: “Tartarite”
- `DeepCoreMiner` y `Comet`: “Meium”
- `StripMiner` y `Asteroid`: “Flavium”
- `DeepCoreMiner` y `Asteroid`: “Dragonite”

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

Probablemente necesite dos métodos beMined... Tomarían su parámetro como puntero no-const, y ambos serían const.

No intente deducir el valor de retorno del asteroide a través de getName(). TIENE que utilizar los TIPOS y el POLIMORFISMO. Cualquier otro tipo de enfoque (typeid, dynamic_cast, getName, etc.) le supondrá un -42. (Sí, incluso si cree que puede salirse con la suya. Porque no, no puede.)

Piense. No es tan difícil como parece. Ahora que nuestros juguetes están listos, contrúyase una nave simpática para ir a minar 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, no más.
- 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.