# Seminario #3: Herencia y Genericidad. C++

## 1) Diseñar una jerarquía de clases que permita modelar las unidades y personajes del universo de Starcraft.

En este videojuego existen 3 razas principales: Terran, Protoss y Zerg.

Todas las unidades de cualquier raza tienen 3 valores enteros:

- la vida
- índice de ataque
- índice de defensa.

Los Protoss además poseen dos valores adicionales:

- Un valor que representa su escudo de protección.
- Un valor que representa cuánto escudo se regenera en un segundo.

Los Zerg por otro lado tienen:

- Un valor extra que representa cuánta vida regenera en un segundo.

```
class Unit{
public:
    int _life, _attack, _defense;
    std::string _name;
protected:
    Unit(int life, int attack, int defense, std::string&& name)
            :_life(life), _attack(attack), _defense(defense), _name(name){}
public:
    void print()
    {
        std::cout << _name << std::endl;
    }
};
```

```
class Zerg : public virtual Unit
{
    int _life_regen;
protected:
    Zerg(int life, int attack, int defense, std::string&& name, int life_regen)
            : Unit(life, attack, defense, static_cast<std::string &&>(name)),
            _life_regen(life_regen){}
};

class Protoss : public virtual Unit
{
    int _shield;
    int _shield_regen;
protected:
    Protoss(int life, int attack, int defense, std::string&& name, int shield,
    int shield_regen)
            :Unit(life, attack, defense, static_cast<std::string &&>(name)),
            _shield(shield), _shield_regen(shield_regen){}
};

class Terran : public virtual Unit {
protected:
    Terran(int life,int attack,int defense, std::string&& name)
            : Unit(life, attack, defense, static_cast<std::string &&>(name)){}
};
```

Nótese que el constructor de la clase Unit está declarado como protected esto significa que no se podrá crear una instancia de esta clase.

## Clases y Herencia:

Una clase es un tipo declarado por el usuario, estas clases pueden ser de distintos tipos en dependencia de la manera en que se declaren (Class, Struct o Union, que es un tipo especial de clase). Estos tipos declarados son indistinguibles en C++ excepto por el modo de acceso y la herencia por defecto de ambos.

La sintaxis para la declaración de una clase es la siguiente:

*Class-Key  Attributes  Class-Name  Base-Clause  { Member-Specification }*

Class-Key:

Ambos Class o Struct pueden ser usados. El uso de la palabra clave Union es una definición de tipo Union, la cual es un tipo especial de clase que puede mantener activo solo uno de sus miembros a la vez.

Attributes:

Secuencia opcional de cualquier número de atributos. Los atributos pueden ser simples, atributos que incluyan un namespace, con argumentos o con namespace y una lista de argumentos. Entre los atributos puede encontrarse alignas, el cual declara el tamaño especifico que va a reservar el tipo declarado. A partir de C++11.

Class-Name:

El nombre de la clase que se va a definir. Opcionalmente puede seguir esta declaración la palabra clave final (desde C++11), la cual indica que esta clase no puede ser heredada.

Base-Clause:

Lista opcional de una o mas clases padres y el tipo de herencia para cada una.

{Member-Specification}

Lista con la declaración de los miembros de una clase.

## Herencia Múltiple:

Al ser C++ un lenguaje que permite la herencia múltiple, cualquier tipo de clase declarado puede derivarse de una o mas clases base, las cuales, a su vez, pueden derivarse de sus propias clases base formando una jerarquía de clases.

Puede surgir una pregunta: *¿Es necesaria la herencia múltiple?*

La respuesta es: depende, la herencia múltiple no es necesaria en el sentido estricto de la palabra, puede construirse de manera equivalente con la herencia simple y el uso de interfaces, el caso es que puede ser útil en muchos casos por lo que es a consideración del programador si la prefiere en la construcción de su aplicación o prefiere escoger un lenguaje con otro tipo de herencia. Por lo tanto, la cuestión no sería si es necesaria o no, la pregunta correcta sería si es útil en el ámbito que se esté trabajando.

La sintaxis correcta para la declaración de las clases bases dentro de Base-Clause consiste en el carácter ‘:’ y luego de esto separado por coma todos los especificadores de clases escritos de la siguiente forma:

*Attribute Access-Specifier Virtual-Specifier Class*

Attribute:

Secuencia opcional de atributos (Al igual que en la declaración de la clase). A partir de C++11. Opcional

Access-Specifier:

Puede ser public, private o protected. Opcional

Virtual-Specifier:

La palabra clave virtual. Opcional

Class:

La clase de la cual se va a heredar.

## 1.b) ¿Cuál es el tipo de herencia por defecto de C++?

Modificadores de acceso en las clases:

Los modificadores de acceso en la herencia de las clases definen la accesibilidad de los miembros heredados. Cuando una clase hereda de otra, los miembros de la clase base se convierten en miembros de la clase derivada. El estado de acceso de los miembros de la clase base dentro de la clase derivada es determinado por el especificador de acceso usado para heredar la clase base.

Si se omite en la declaración el modificador de acceso, el tipo de **herencia por defecto** es privada en las clases y pública para los struct.

¿Qué representan en la herencia los modificadores de acceso?

Herencia pública (public):

Cuando una clase utiliza public como modificador de acceso en la herencia de una clase base, todos los miembros públicos de la clase base son accesibles como miembros públicos de la clase derivada y todos los miembros protegidos de la clase base son accesibles como miembros protegidos de la clase derivada. Los campos o funciones privadas de la clase base nunca son accesibles a menos que se utilicen funciones friend.

Herencia protegida (protected):

Cuando una clase utiliza protected como modificador de acceso en la herencia de una clase base todos los miembros públicos y protegidos de una clase base son accesibles como miembros protegidos en la clase derivada. La herencia protegida puede ser útil para mantener un polimorfismo controlado. Los campos o funciones privadas de la clase base nunca son accesibles a menos que se utilicen funciones friend.

Herencia privada (private):

Cuando una clase utiliza private como modificador de acceso en la herencia de una clase base todos los miembros públicos y protegidos de una clase base son accesibles como miembros privados en la clase derivada. Los campos o funciones privadas de la clase base nunca son accesibles a menos que se utilicen funciones friend.

Ejemplo de cómo funciona la herencia para distintos modificadores de acceso:

```
class Base
{
public:
    int public_field;
private:
    int private_field;
protected:
    int protected_field;
};

class Pub : public Base
{
public:
    Pub(){
        public_field = 1;       //ok: public_field fue heredado como publico
        protected_field = 2;    //ok: protected_field fue heredado como protegido
        private_field = 3;      //not ok: private_field no es accesible en la clase derivada
    }
};


class Priv : private Base
{
public:
    Priv(){
        public_field = 1;      //ok: public_field es ahora privado en la clase derivada
        protected_field = 2;   //ok: protected_field es ahora privado en la clase derivada
        private_field = 3;     //not ok: private_field es inaccesible dentro de la clase derivada
    }
};
```

```
class Prot : protected Base
{
public:
    Prot(){
        public_field = 1;     //ok: public_field es ahora protegido en la clase derivada
        protected_field = 2;  //ok: protected_field es protegido en la clase derivada
        private_field = 3;    //not ok: private_field no es accesible dentro de la clase derivada

    }
};
```

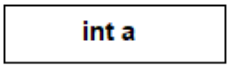
## 1.c) ¿Cómo se representa en memoria la herencia en C++?

Herencia Simple:

Un objeto de tipo clase en C++ es representado por una región continua en la memoria. Un puntero a una instancia de una clase apunta el primer byte de esa región de memoria. Sea una clase A:

```
class A
{
  int a;
};
```

Representación en memoria




En la memoria correspondiente a una instancia de A solo aparecerá el entero especificado por el usuario.

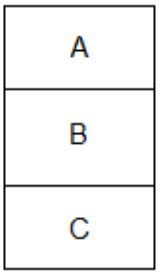
Herencia Múltiple:

De igual manera se designa una región continua en memoria donde se almacena una instancia de cada una de las clases de las que hereda el objeto. Por ejemplo:

```
class A { /*body*/ };
class B { /*body*/ };

class C : A , B { /*body*/ };
```

Representación en memoria



2) Además de estas 3 razas, existen 2 adicionales que son mezclas de las anteriores:

- Terran Infestados (Terran-Zerg)
- Los Híbridos (Protoss-Zerg)

2.a) Modele esto utilizando herencia múltiple y virtual.

```
class Infected_Terran : Terran, Zerg
{
    Infected_Terran(int life, int attack, int defense, std::string&& name, int life_regen)
            : Unit(life, attack, defense, static_cast<std::string &&>(name)),
              Zerg(life, attack, defense, static_cast<std::string &&>(name), life_regen),
              Terran(life, attack, defense, static_cast<std::string &&>(name)){}
};
```

```
class Hybrid : Protoss, Zerg
{
public:
    Hybrid(int life, int attack, int defense, std::string&& name, int shield, int shield_regen, int life_regen)
            : Unit(life, attack, defense, static_cast<std::string &&>(name)),
              Protoss(life, attack, defense, static_cast<std::string &&>(name), shield, shield_regen),
              Zerg(life, attack, defense, static_cast<std::string &&>(name), life_regen){}
};
```

La palabra clave virtual puede usarse en dos contextos, como modificador de una función perteneciente a una clase o como modificador de las clases bases heredadas.

Modificador de función:

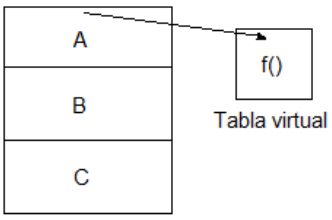
Las funciones virtuales son aquellas cuyo comportamiento se puede sobrescribir en las clases derivadas. Estas funciones en las clases derivadas son virtuales también, incluso si la palabra reservada virtual no fue usada en su declaración y sobrescriben la función correspondiente en la clase base aun si la palabra override no se usó en la declaración.

Cada instancia de una clase posee al inicio de su representación en memoria un puntero virtual a una tabla virtual donde se guardan las definiciones de los métodos virtuales usados por ella.

Por ejemplo, si se tiene:

```
class A { virtual void f(); };
class B : public A { virtual void f(); };
class C : public B { /*body*/ };
```

Representación en memoria



Las instancias de A, B, C apuntan a la función f declarada en B.

Modificador de clase heredada:

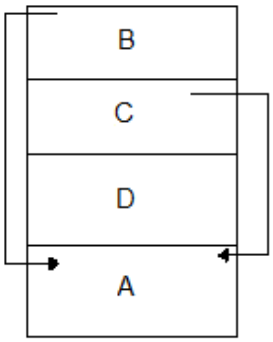
Por cada clase base distinta que es especificada virtual, el objeto general más derivado posee una sola instancia de dicha clase, incluso si la clase aparece varias veces en la jerarquía de herencia.

En memoria, por cada clase A que hereda virtualmente de otra clase B, se mantiene un puntero virtual hacia la clase B antes de la representación de dicha clase A.

Por ejemplo:

```
class A { /*body*/ };
class B : virtual A { /*body*/ };
class C : virtual A { /*body*/ };
class D : B, C { /*body*/ };
```

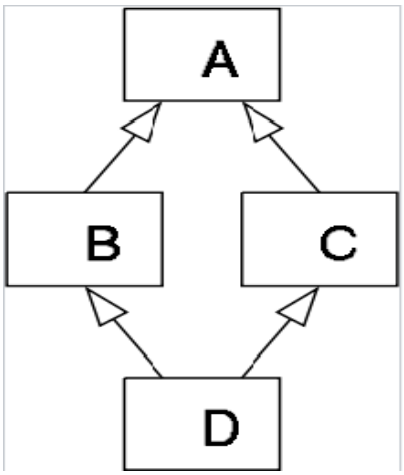
Representación en memoria




La instancia de A usada por los objetos de tipo B y C es la misma.

## 2.b) ¿Qué problemas trae la herencia múltiple con respecto a la representación en memoria de la herencia?

En los lenguajes de programación orientada a objetos, el **problema del diamante** es una ambigüedad que surge cuando dos clases B y C heredan de A, y la clase D hereda de B y C. Si un método en D llama a un método definido en A, ¿por qué clase lo hereda, B o C?




En nuestro caso este problema se presenta ya que la clase **Unit** es heredada por **Hybrid** dos veces a través de la clase **Protoss** y **Zerg.** Esto significa que un objeto de tipo **Hybrid** tendrá dos atributos y funciones por cada uno de los de **Unit**, correspondientes a las instancias de Unit que se encuentran en Zerg y Protoss respectivamente. Esto hace que el compilador no pueda diferenciar cuál atributo o función tomar ante un llamado.

La clase Hybrid hereda de las clases: Protoss y Zerg que, a su vez, ambas heredan de la clase Unit; por lo tanto, en las propiedades **name, life, attack, defense** y en el método **print** el compilador no puede diferenciar entre estas propiedades de la clase Protoss o Zerg.

La solución a este problema es el uso de la herencia virtual en las clases, por ejemplo, al hacer a Protoss y Zerg heredar virtualmente de Unit, un objeto Hybrid poseerá una sola instancia de la clase Unit, eliminando las ambigüedades. Aun así, la herencia virtual puede traer un uso excesivo de la memoria debido a que la cantidad de punteros que se creen para una clase en específico puede llegar a ser muy grande.

3) A partir de estas clases cree una que represente las unidades con poderes. Esta clase debe ser genérica en 3 parámetros que representan los poderes que se les puede asignar. Se deben restringir los parámetros genéricos para que sólo acepten clases concretas que hereden de Power. Sólo el primer parámetro genérico debe ser obligatorio, los demás pueden no asignarse.

```cpp
template<class P1, class P2 = nullptr_t , class P3 = nullptr_t >
class Powered_Unit : public Mana_Unit
{
    static_assert(std::is_base_of<Power, P1>::value and not std::is_abstract<P1>::value,
                  "P1 must inherit from Power and be a concrete type.");
    static_assert(std::__is_nullptr_t<P2>::value or (std::is_base_of<Power, P2>::value and not std::is_abstract<P2>::value)
                  "P2 must inherit from Power and be a concrete type.");
    static_assert(std::__is_nullptr_t<P3>::value or (std::is_base_of<Power, P3>::value and not std::is_abstract<P3>::value)
                  "P3 must inherit from Power and be a concrete type.");
public:
    P1 _power1;
    P2 _power2;
    P3 _power3;

    Powered_Unit(int mana, int genManaPerSecond)
          : Mana_Unit(mana, genManaPerSecond){}

    virtual void Cast_Power1() = 0;
    virtual void Cast_Power2() = 0;
    virtual void Cast_Power3() = 0;
};
```

¿Qué son los templates?

Los templates constituyen la base para la programación genérica en C++, debido a que este es un lenguaje fuertemente tipado requiere que todas las variables tengan un tipo específico, ya sea declarado explícitamente o deducido por el compilador. Sin embargo, el comportamiento de muchas estructuras de datos y algoritmos no varían con respecto al tipo que están operando. El esquema de templates permite definir las operaciones de una clase o función y dejan al usuario especificar sobre qué tipos concretos estas operaciones deben trabajar.

## 3.a) Genericidad con templates.

Template Parameters:
Los parámetros en los templates definen qué elementos dentro de la definición realizada quedan indeterminados. La definición más básica de los parámetros de un template se realiza de la siguiente forma:

```
//template< comma-separated-list-of-parameters > i.e.
template<typename T> //template<class T> is valid too

//some class or function definition...
```

¿Existe alguna diferencia entre `template<typename T>` y `template<class T>`?
No, para el compilador estas definiciones son equivalentes, sin embargo, algunos programadores utilizan `class` para especificar que el template ha sido pensado para ser utilizado con clases y no con otro tipo de parámetros.
Los parámetros no tienen por qué ser tipos, también pueden ser valores e incluso otros templates, al igual que al parametrizar templates con tipos se definen los valores aceptados, sin embargo, los valores oridnarios deben ser especificados explícitamente a la hora de instanciar el template.

```
template<typename T, size_t L>
class MyArray {
    T arr[L];
public:
    MyArray() {...}
}
MyArray<int, 10> arr; //An array for ten ints

//templates as parameters
template<typename T, template<typename U, int I> class Arr>
class SomeClass{...}
```

Function Templates:
Los function templates proveen un comportamiento funcional que puede ser utilizado por diferentes tipos, estos representan una familia de funciones para las cuales algunos elementos de su definición son indeterminados, estos elementos son parametrizados y especificados durante la definición del template.

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

En este ejemplo definimos una función máximo genérica la cual puede ser utilizada por cualquier tipo cuyos elementos puedan ser comparados.
Los function templates al igual que las funciones comunes pueden ser sobrecargadas, de esta manera se pueden tener diferentes definiciones de funciones con el mismo nombre y el compilador de C++ se encargará de elegir qué función llamar.

```
template<typename T1, typename T2>
auto max(T1 a, T2 b) {
    return b < a ? a : b;
}
max('a', 1.0)  //calls max<char, double>
```

Al trabajar con sobrecarga de funciones pueden producirse errores si el compilador no puede determinar cuál de todas las posibles definiciones de una función llamar.

```
max(1.0, 2.0)  //calls max<double> or calls max<double, double> ?
```

Desde C++11, se puede utilizar constexpr para utilizar código en el cómputo de algunos valores en tiempo de compilación.

```
template<typename T1, typename T2>
constexpr auto max(T1 a, T2 b) {
    return b < a ? a : b;
}
int a[::max(sizeof(char), 1000u)];
```

Class Templates:
Al igual que con funciones las clases pueden ser parametrizadas con uno o más tipos.

```
#include <vector>
template<typename T>
class Stack{
private:
    std::vector<T> elems; //elements
public:
    //stack methods...
}
```

Para definir una función miembro de una clase parametrizada con templates es necesario hacer uso del template en la definición de las funciones.

```
template<typename T>
void Stack<T>::push (T const& elem){
    elems.push_back(elem);
}
```

Se debe de tomar otras consideraciones adicionales al trabajar con templates, véase el siguiente código:

```
template<typename C>
void f(const C& container){
    C::const_iterator * x;
    ...
}
```

Este código aunque aparenta declarar un puntero al campo C::const_iterator no es tan obvio para el compilador debido a que este carece de información sobre el tipo C, por tanto le es imposible determinar si el nombre const_iterator el cual depende de C, a este tipo de nombre se les denomina *nested dependent type name*, se refiere a un tipo o no. C ++ por convenio asume que no es un tipo lo que puede llevar a serios errores. Para indicar que un nombre es un tipo dentro de un tipo utilizado como parámetro de template se debe utilizar la palabra clave typename.

```
Typename C::const_iterator * x;
```

## 3.b) Valores por defecto de los templates.

Los parámetros de los templates pueden tener valores definidos por defecto, el valor especificado puede ser de cualquier tipo e incluso puede depender de parámetros previamente definidos, si un parámetro tiene valor por defecto esto permite dejarlo sin especificar al utilizar el template:

```
//template for a vector
template<class T, class Allocator = allocator<T>> class vector {...};
//The template parameter Allocator allow us to define a custom memory allocator o
r to use a predefined allocator for the type of the array's elements
auto v = vector<int>(...)
```

## 3.c) Restricciones sobre los templates.

C++ permite el uso parcial de los class templates, es decir, el tipo concreto pasado al parámetro de la clase no tiene que soportar todas las operaciones que se definen sobre él en la clase, solo aquellas utilizadas en tiempo de ejecución. Esto es útil para no tener que crear una nueva clase más restringida cuando un tipo no soporte todas las operaciones definidas en la clase.

A modo de ejemplo podemos suponer que la clase template<typename T> class Stack{...} implementa un método print() el cual escribe los valores de la pila en la consola, de no poder utilizar parcialmente la clase tendríamos que crear una nueva clase Stack para aquellos tipos que no implementen el operador << o se pudiese implementar una función no miembro que reciba un Stack e imprima su contenido, sin embargo el uso parcial permite mantener la encapsulación de la funcionalidad dentro de la clase Stack. Sin embargo, esto plantea un nuevo problema, dado que distintos tipos pueden soportar distintas funcionalidades dentro de una clase como saber si uno de estos tipos soporta cierta funcionalidad o asegurar que los tipos pasados como parámetros implementen las operaciones necesarias para que la clase pueda ser instanciada.

Para resolver este problema C++ implementa los llamados *concepts* los cuales describen un conjunto de requerimientos para un tipo, y desde C++11 podemos utilizar static_assert para verificar que los tipos cumplan ciertas restricciones.

```
class BaseClass {}

template<typename T>
class ExampleClass{
    static_assert(std::is_base_of<BaseClass, T>::value, "type parameter of this c
lass must derive from BaseClass");
...
}
```

Cuando un template es llamado, el compilador remplaza cada instancia de T por el tipo concreto del argumento el cual puede ser definido por el programador o inferido por el compilador, por lo que los templates son compilados por cada tipo (lista de tipos) por los que son usados. Este proceso es llamado *instantiation* y resulta en una instancia de un template.

Los templates son compilados en dos fases:

1. Antes de realizar la instanciación del template (tiempo de definición) el código de la definición es revisado ignorando los parámetros del template, se detectan errores sintácticos y errores semánticos que no dependan de los parámetros.

2. En tiempo de instanciación el código del template es nuevamente revisado dado que el conocimiento de los tipos concretos de los parámetros del template permite al compilador verificar que el código dependiente de estos sea válido.

¿Como determina el compilador que tipos pueden ser utilizados en un template?

En el esquema de templates de C++ las interfaces quedan definidas implícitamente de acuerdo a las funcionalidades asociadas al tipo dentro de las expresiones que lo referencian en el template.

```
template<typename T>
int compare(T x, T y){
    if (x > y) return 1;
    if (x < y) return -1;
    return 0;
}
```

Se infiere a partir del uso de los argumentos x y y que el tipo T debe de implementar los operadores de comparación dentro de su mismo tipo.

## 4) Cada raza tiene unidades únicas o héroes, los cuales pueden poseer poderes especiales y de los cuales sólo puede existir una instancia.

### 4.a) Explicar el patrón Singleton.

¿Qué es un patrón de diseño?

Un patrón de diseño es una descripción de clases y objetos comunicándose entre sí para resolver un problema de diseño general en un contexto particular.

Los patrones de diseño se encuentran distribuidos en 3 grupos: Patrones Creacionales, Estructurales y de Comportamiento.

El patrón Singleton se encuentra ubicado en el grupo de patrones creacionales. Como su nombre lo sugiere, del inglés "*single*" que significa "*único*", trata de diseñar clases que solo pueden ser instanciadas una sola vez y proporciona un punto de acceso global a ella, es decir que, de esa clase, durante toda la ejecución del programa, únicamente podrá ser creado un objeto, pero podemos interactuar con el mismo, nos permite tener el control sobre la asignación y destrucción del objeto. El Singleton es útil, por ejemplo, cuando una clase es diseñada para representar un dispositivo único dentro de un programa, ejemplo el teclado y el ratón por mencionar algunos, también es requerido en diseños de clases que interactúan con todas las demás como un recurso común, algo así como una clase de ámbito global.

Para diseñar una clase singleton hay algunos puntos importantes a tomar en cuenta:

1- Asegurar la creación de un único objeto, ya que es el propósito general de la clase singleton, el mecanismo de creación de objetos debe ser modificado de tal manera que se utilice solo una vez el constructor, para esto dejamos que la clase misma controle su propio constructor y que ninguna otra entidad tenga acceso a este, para ello declaramos el constructor como privado y programar lo necesario para que este solo sea usado una vez en todo el programa.
2- Luego de resolver el problema de la unicidad de dicho objeto surge la interrogante de como interactuar con el mismo, el truco está en proporcionar una interfaz desde la clase para que las demás entidades puedan interactuar con este objeto, esto se resuelve creando un método de clase público y estático para que pueda ser llamado desde afuera de la clase sin necesidad de haber creado un objeto de esta.
3- Al asegurarse de que la clase controla el constructor y que provee una interfaz para interactuar con su único objeto, resta pensar en cómo existe y se maneja esta instancia dentro de la clase, esto recae en usar una referencia o apuntador como la variable que al final contendrá la memoria que representa al objeto, esta variable debe ser un atributo perteneciente a la clase, no a los objetos de dicha clase: esto es necesario porque, al no tener un objeto inicialmente, se requiere usar un atributo accesible desde la clase, la variable debe ser visible únicamente dentro de la clase, aunque esta es una característica fundamental de la programación orientada a objetos pero no está de más recordar que hay que aplicar encapsulamiento para proteger los atributos de la clase. Solo queda discutir sobre la interfaz con el exterior y el uso interno del constructor. Para empezar dicha interfaz será usada por las entidades que quieran interactuar con el singleton, entonces cada vez que se llame a este método, dentro de la clase se deberá acceder al objeto y nos encontramos con dos posibilidades que el objeto no esté creado y debe ser construido y guardado en el atributo de referencia para finalmente retornar esa referencia, esta será la primera vez que se utiliza el constructor y si el objeto ya existe basta con retornar la referencia existente.

```
//Singleton
class Marine_Hero : public Terran, public Powered_Unit<Machine_Gun>
{
protected:
    static Marine_Hero *instance;

public:
    static Marine_Hero* Create_Instance(int life, int attack, int defense,
    std::string&& name, int mana, int genMana, Machine_Gun power)
    {
        if(instance != nullptr) return instance;
        instance = new Marine_Hero(life, attack, defense, static_cast<std::string
         &&>(name), mana, genMana, power);
```

```
        return instance;
    }
    static Marine_Hero* Get_Instance()
    {
        if(instance == nullptr)
                static_assert(true, "An instance of this class has not been
                created");
        return instance;
    }
    void Cast_Power1(){ std::cout<<"Casting power 1"; }
    void Cast_Power2(){ std::cout<<"There is not power 2"; }
    void Cast_Power3(){ std::cout<<"There is not power 3"; }

private:
    Marine_Hero(int life, int attack, int defense, std::string&& name, int mana,
     int genMana, Machine_Gun power):
            Terran(life, attack, defense, static_cast<std::string &&>(name)),
            Powered_Unit<Machine_Gun>(mana, genMana),
            Unit(life, attack, defense, static_cast<std::string &&>(name))
            { Powered_Unit::_power1 = power; }
};
Marine_Hero *Marine_Hero::instance = nullptr;
```

## 5) Implemente algunos ejemplos de poderes, unidades y héroes.

## 5.a) Explicar la especialización de templates.

En ocasiones no es conveniente tener la misma implementación de una clase genérica para todos los posibles tipos que soporta, por ejemplo, cuando se quiere optimizar la implementación o resolver ciertos comportamientos inesperados para ciertos tipos. Para resolver esta necesidad C++ permite la especialización de class templates. Sin embargo, debe considerarse que si se especializa un class template deben de especializarse todas sus funciones miembros.

```
//Specialization of the class Stack for string
template<>
class Stack<std::string>{
    ...
    template<>
    void Stack<std::string>::push (std::string const& elem){
        elems.push_back(elem);
```

```
    }
    ...
}
```

La posibilidad de especializar class templates también trae un nuevo problema, véase el siguiente ejemplo:

```
template<typename T>
class BaseClass{
    ...
    void f() {}
    ...
}
template<typename T>
class DerivedClass: BaseClass<T>{
    ...
    //using the inherited function f()
    void g(){
        ...
        f(); //here we will get a compilation error
        ...
    }
    ...
}
```

Este código provoca un error de compilación debido a que el compilador reconoce que la clase `BaseClass` puede estar especializada para algún tipo concreto y por tanto no puede asegurarse que implemente el método f( ), para solucionar esto se referencia al método f( ) utilizando el modificador this para indicarle al compilador que asuma que el método va a ser heredado.

Los class templates pueden ser también parcialmente especializados para poder proveer implementaciones específicas donde puede ser conveniente o incluso necesario que algunos parámetros sigan siendo especificados por el usuario.

```
//Specialization of the class Stack for pointers
template<typename T>
class Stack<T*>{
    ...
```

```
}


//Especializacion
template<>
class Powered_Terran<Get_Resources> : public Terran, Powered_Unit<Get_Resources>
{
public:
    Powered_Terran(int life, int attack, int defense, std::string&& name, int mana, int genMana) :
            Terran(life, attack, defense, static_cast<std::string &&>(name)),
            Powered_Unit<Get_Resources>(mana, genMana),
            Unit(life, attack, defense, static_cast<std::string &&>(name)) {}
    void Talk() { std::cout<<"Ready for your commmand"; }
    void Cast_Power1(){ std::cout<<"Casting power 1"; }
    void Cast_Power2(){ std::cout<<"There is not power 2"; }
    void Cast_Power3(){ std::cout<<"There is not power 3"; }
};
```