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 regeneran 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;
public:
    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;
public:
    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 {
public:
    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 esta 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 en 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 seria si es necesaria o no, la pregunta correcta seria si es útil en el ámbito que se este 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 protected 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 como 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

    }
};
```

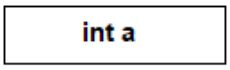
1.c) ¿Cómo se representa en memoria la herencia en C++?

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.

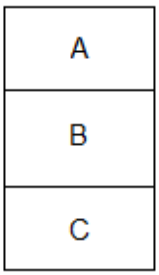
Herencia Múltiple:

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:

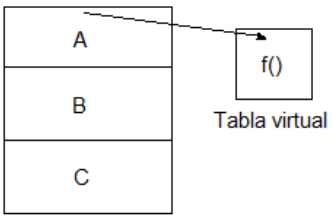
Las funciones virtuales son aquellas cuyo comportamiento se puede sobreescribir en las clases derivadas. Estas funciones en las clases derivadas son virtuales también, incluso si la palabra reservada virtual no fue usada en su declaración y sobreescriben la función correspondiente en la clase base aun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

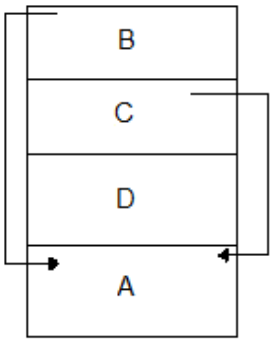
Por cada clase base distinta que es especificada virtual, el objeto general mas derivado posee una sola instancia de dicha clase, incluso si la clase aparece varias veces en la jerarquía de herencia.

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*/ };
```

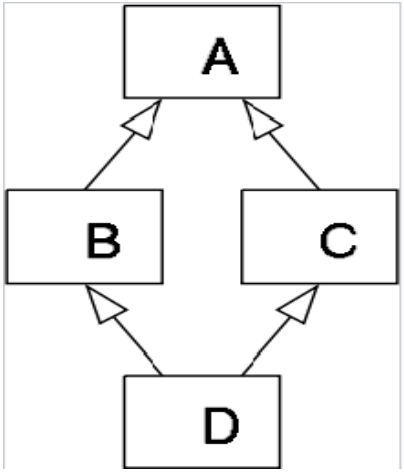
Representación en memoria



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 cual 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 especifico 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.

```
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;
};
```

¿Que 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.

a.) Genericidad con templates.

Un template es una construcción que genera un tipo o una función en tiempo de compilación basándose en los argumentos que el usuario provee a través de los parámetros del template.

```
template <typename T>
T minimun(const T& lhs, const T& rhs)
{
   return lhs < rhs ? lhs : rhs;
}
```

No existen límites prácticos en cuanto a la cantidad de parámetros que puede tener un template. Sin embargo, existe una definición mas comoda para declarar que un template recibe un numero arbitrario de argumentos.

```
template <typename ...arguments> class example_class { }
```

Los parámetros de los templates pueden ser *type*, *non-type* o *template*.

Las siguientes definiciones de templates son equivalentes:

```
template <typename T> ...
template <class T> ...
```

Sin embargo algunos programadores optan por la convención de utilizar class para especificar que el template esta diseñado para ser utilizado con clases.

En C++ el esquema de templates se basa en interfaces implícitas y polimorfismo en tiempo de compilación, véase el siguiente ejemplo:

```
class Widget { }
template <typename T>
void do_processing(T& w)
{
  if(w.size() > 10 && w != some_widget)
  {
    T temp(w);
    temp.normalize();
    temp.swap(w);
  }
}
```

b. Valores por defecto a los templates.

Los templates pueden tener argumentos por defecto. Cuando un template tiene un argumento por defecto este puede ser dejado sin especificar cuando se utiliza dicho template estos se definen de la siguiente forma:

```
class A { ... }

template<class C, class T = A> ...
```

esto indica que en caso de no especificar el segundo argumento se utilizara el tipo A, ademas en caso de tener multiples argumentos en un mismo template todos los argumentos después del primer argumento con valor por defecto deben tener definidos valores por defecto también.

c. Restricciones sobre los templates.

//

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

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 este 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"; }

protected:
    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.

a) Explicar la especialización de templates.

La especialización de los templates se realiza de la siguiente forma:

```
template<>
class Base_class <c_type>
{
```

```
    void another_func() { }
}
```

Esto le indica al compilador que cuando utilizamos el template
template <typename C> class Base_class {...}

con argumento c_type este será la definición a usar. Aquí el template Base_class esta especializado para el tipo c_type.

Por esto C++ reconoce que los templates pueden estar especializados y pueden no ofrecer las mismas interfaces que en su definición inicial. Para solucionar este problema es recomendado utilizar el modificador this delante de los métodos de la clase base.

```
template <typename C>
class Derived_class: Base_class<C>
{
    void g()
    {
        this->f();
        //...
    }
}
```

Esto le indica al compilador que Base_class implementa el método f y que Derived_class hereda de este método.

```
//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"; }
};
```