**Visibilidad respecto a una clase:**

En la declaración de una clase existe la posibilidad de declarar los modificadores de visibilidad de los miembros de la misma (**public**, **protected**, **private**).Dependiendo del modificador utilizado se puede acceder a los miembros de la clase desde distintos lugares del programa

**Public**: puede ser accedido desde los miembros o no miembros de la clase programa

**Protected**: puede ser accedido solo desde los propios miembros de la clase y de las clases descendientes

**Private**: puede ser accedido solamente por los miembros de la propia clase

**Visibilidad respecto a la herencia**:

El proceso de herencia puede efectuarse de dos formas distintas: siendo la clase base ***public*** o ***private*** para la clase derivada. En el caso de que la clase base sea ***public*** para la clase derivada, esta hereda los miembros ***public*** y ***protected*** de la clase base como miembros ***public*** y ***protected,*** respectivamente. Por el contrario, si la clase base es ***private*** para la clase derivada, esta hereda todos los datos de la clase base como ***private***.

Para indicar que una clase derivada de otra es necesario especificarlo en la declaración de la clase derivada, especificando también modificador de visibilidad en que deriva de su clase base.

**Sintaxis:**

**Class Derivada: [**virtual**] [**public/private/Protected**] Clase\_Base**

De no especificar el modificador de visibilidad hay que tener en cuenta que la herencia predeterminada es la misma que la visibilidad predeterminada de los miembros: *public* si usa la palabra clave *struct*, y *private* para la palabra clave *class*.

Incluso es posible tener una *class* derivada de una *struct* (o viceversa). En este caso, la herencia predeterminada está controlada por el hijo, por lo que una *struct* que se deriva de una *class* se establecerá de forma predeterminada como herencia publica, y una *class* que se deriva de una *struct* tendrá herencia privada de forma predeterminada.

**Herencia**

Una clase simple se representa en memoria como la concatenación de todas sus propiedades y métodos en orden de inicialización.

Una clase derivada se representa en memoria como una concatenación de los elementos de sus clases bases (En caso de ser herencia simple, una sola clase base) y luego los elementos de ella misma. Por cada clase base se almacenan los miembros que se heredan de ella, en el mismo orden. Si alguna clase base, o la derivada poseen algún método virtual, por cada uno de estos se almacena un puntero a una estructura que contiene la información de los métodos virtuales (**virtual method table** o **vTable**) que se encarga gestionar los override a los métodos virtuales.

Supongamos que tenemos las clases First con los métodos virtuales A, B y Second que hereda de First y hace override al método A, y tiene un método Virtual propio C. La siguiente imagen muestra cómo queda registrado en memoria ambas clases.

|  |
| --- |
| First |
| A |
| B |

|  |
| --- |
| First  vTable |
| A |
| B |

|  |
| --- |
| Methods Location |
| First: A |
| First: B |
| Second: A |
| Second: C |

|  |
| --- |
| Second |
| First: A |
| First: B |
| C |

|  |
| --- |
| Second  vTable |
| A |
| B |
| C |

El orden en que se representan los datos dentro de la clase es susceptible a la implementación del compilador.

**Ejemplo De Herencia Simple**

En el proyecto se tienen las clases Unit y Zerg, donde Zerg hereda de Unit.

Unit está compuesta por las propiedades: attack, deffense, life y name.

La clase Zerg al heredar de Unit, posee las mismas propiedades que Unit, además de regLife que indica cuanta vida regenera por segundo.

En este ejemplo de herencia simple, el compilador de C++ guarda las propiedades en orden de inicialización, y luego de la clase que hereda. La clase Zerg en memoria queda representada de la siguiente manera:

|  |
| --- |
| Zerg: |
| Unit::attack |
| Unit::deffense |
| Unit::life |
| Unit::name |
| regLife |

**Ejemplo Herencia Múltiple**

Trabajando sobre el ejemplo anterior agreguemos las clases Protoss e Hibrido, donde Protoss hereda de Unit e Hibrido hereda de Protoss y Zerg. Su representación en memoria seria la siguiente:

UNIT

ZERG

PROTOSS

UNIT

HIBRIDO

|  |
| --- |
| Protoss: |
| Unit::attack |
| Unit::deffense |
| Unit::life |
| Unit::name |
| Shield |
| regShield |

|  |
| --- |
| Hibrido: |
| Zerg::Unit::attack |
| Zerg::Unit::deffense |
| Zerg::Unit::life |
| Zerg::Unit::name |
| Zerg::regLife |
| Protoss::Unit::attack |
| Protoss ::Unit::deffense |
| Protoss ::Unit::life |
| Protoss ::Unit::name |
| Protoss::Shield |
| Protoss::regShield |

Aquí podemos ver como en la clase Hibrido tiene repetida varias propiedades de la clase Unit. Esto puede causar problemas de ambigüedad al querer llamar a una propiedad con un mismo nombre. Por ejemplo, si hacemos

Hibrido h = new Hibrido (…)

P = h -> attack

En este código el compilador devuelve error de ambigüedad, pues no sabe a qué propiedad se refiere verdaderamente. Para resolver este conflicto realizamos un casteo:

P = h -> (\*Zerg) attack <= Igual funciona con (\*Protoss)

Herencia virtual también soluciona este problema.

**Herencia Virtual**

Con herencia virtual evitamos la repetición de elementos en la representación de memoria de nuestra clase derivada con el costo adicionado de tener una tabla virtual (la **vTable** mencionada anteriormente).

Las tablas de funciones virtuales son arreglos que contienen punteros a funciones o propiedades. Cada clase que contenga funciones virtuales tiene uno o más apuntadores asociados a tablas virtuales. Para cada función virtual dentro de una clase, ya sea heredada o definida en dicha clase, existe en la ***vTable***una entrada que contiene un puntero a la función virtual que ejecutará un objeto de ese tipo, esa función puede ser la definida o redefinida en la clase, o puede ser la función declarada como virtual en una clase base, que no fue redefinida.

Usando clases virtuales la nueva representación en memoria de la clase Hibrido es la siguiente

UNIT

PROTOSS

HIBRIDO

ZERG

|  |
| --- |
| Hibrido: |
| Vptr:Zerg |
| Zerg::regLife |
| Vptr:Protoss |
| Protoss::Shield |
| Protoss::regShield |
| Unit::attack |
| Unit::deffense |
| Unit::life |
| Unit::name |

|  |
| --- |
| VTable: |
| Virtual Base Offset = 8  Offset To Top = 0  TypeInfo for Hibrido |
| Virtual Base Offset = 6  Offset To Top = 2  TypeInfo for Hibrido |

En este caso cuando quisiéramos acceder a alguna propiedad Unit de Hibrido no es necesario hacer un casteo como en el ejemplo de herencia múltiple pues los punteros virtual se aseguran de que si se accede por Zerg o por Protoss, ambos llegan a la misma dirección de memoria que señala a las propiedades Unit.

**Casts**

**En c++ existen los siguientes cast:**

* **cast tipo C**
* **dynamic\_cast**
* **static\_cast**
* **reinterpret\_cast**
* **const\_cast**

Es importante mencionar que excepto el dynamic\_cast el resto se realizan en tiempo de compilación y el dynamic cast se realiza en tiempo de ejecución.

**Cast tipo C**

**La sintaxis de este casting es la siguiente :**

**(NewType)variable**

Cuando se utiliza este lanzamiento, usa uno de los siguientes lanzamientos de c ++ (en orden):

• const\_cast<NewType>(variable)

• static\_cast<NewType>(variable)

• const\_cast<NewType>(static\_cast<const NewType>(variable))

• reinterpret\_cast<const NewType>(variable)

• const\_cast<NewType>(reinterpret\_cast<const NewType>(variable))

**Debido a que este lanzamiento puede dar lugar a reinterpret\_cast no deseados puede dar problemas y a menudo se considera peligroso e incluso una mala practica.**

**Dynamic\_Cast:**

**La sintaxis de este casting es la siguiente :**

**dynamic\_cast<type\*>(&a)**

Se realiza en tiempo de ejecución y devuelve null si no se pudo convertir **&a** en **type\*.** Resulta

especialmente útil con el polimorfismo, ya que permite una navegación a través de la jerarquía de

herencia, puede convertir una clase derivada una clase base o una clase base en una derivada (sólo va a

poderse realizar esta última conversión si los tipos son polimórficos, osea tiene al menos un método

virtual)

**Static\_cast:**

**La sintaxis de este casting es la siguiente :**

**static\_cast<type\*>(&a)**

Este cast trata de convertir de forma segura &a a un type\* y si no resulta devuelve un error en tiempo

de compilación. Puede convertir una clase derivada en una clase base y también puede revertir un implicit cast.

**Const\_cast:**

**La sintaxis de este casting es la siguiente :**

**const\_cast<new type\*>(expression)**

El const\_cast se emplea para trabajar con elementos const y lidiar con el acceso a miembros de la clase

en un método const

**Reinterpret\_cast:**

**La sintaxis de este casting es la siguiente :**

**reinterpret\_cast<new type\*>(memory offset)**

De los casts el reinterpret\_cast es el mas 'loco' ya que reinterpreta la direccion de memoria como el

tipo determinado cuando se llamo al método.

Por ejemplo:

int\* p3 = reinterpret\_cast<int\*>(&a)

**Templates**

Un **Template** es una simple y poderosa herramienta de C++. La idea general es pasar los tipos de datos como parámetro para no tener que rescribir el mismo código para tipos de datos diferente. Un ejemplo de esto sería tener la función sort() y pasarle como parámetros los tipos de datos, pudiendo hacer una ordenacion de diferentes tipos si tener que reescribir la función para cada uno en especifico

**Sintaxis:**

**Template<Typename T,Typename X,......>**

También se puede sustituir **Typename** por **Class** sin existir ninguna diferencia semantica entre ambos

¿C**uál es la diferencia entre la sobrecarga de métodos y la utilización de template?**

Las dos son ejemplos de polimorfismo relacionado con la **POO.** La sobrecarga se utiliza cuando las funciones tienen un comportamiento parecido en cambio los template se utilizan solamente cuando el funcionamiento es idéntico

**¿Cómo funciona? ¿Que está pasando?**

El compilador no genera ningún código cuando ve la función **Template** sino que lo realiza cuando es llamado con argumentos concreto. El compilador lo que realiza es que cada vez que ve el valor genérico asignado, en cualquier parte del código lo remplaza con el tipo concreto (tipo una macro)

Los **Template** pueden ser de funciones, clases o variables. Si al definir una función su encabezado es un **Template** indica que en el código los parámetros será utilizados genéricamente por lo que se pude definir sin tener que especificar el tipo de algunos de sus miembros. Para las clases es similar y todas las funciones podrán utilizar tipos genéricos

**Argumento por defectos:**

En los Template de clases se pueden definir valores por defectos para los parámetros recalcando su tipo. Si no se especifica el valor del parámetro el compilador le asignara el valor por defecto que fue declarado. Para más claridad veamos el siguiente ejemplo.

template<typename T, typename Cont=std::vector<T> >

class Stack {

private:

Cont elems;

public:

void push(T const&);

...

};

template<typename T, typename Cont>

void Stack<T, Cont>::push(T const& elem) {

elems.push\_back(elem);

Como se puede apreciar en la clase Stack<> se puede definir el contenedor que será usado para manejar los elementos como un segundo parámetro de plantilla, usando std::vector como valor por defecto.

**Restricción de parámetros**

A los tipos genéricos se le pude hacer alguna restricción sobre los parámetros controlando así un poco el valor de entrada. Para esto se cuenta con la función static\_assert(A, mensaje) que muestra una excepción si la condición A no se cumple.

**Especialización de los Templates**

La especialización de **Templates**  de clases nos permite especializar un **Template**  de clase para un tipo de datos en particular. Esto permite que el código de un **Template**  se personalice para un valor de argumento especifico logrando así una mayor eficiencia para algún tipo y evitando que se corra el mismo segmento de código para todos los tipos de parámetros.

**Por ejemplo:**

template<class T>

class max

{

T x1;

T x2;

public :

max(T x,T y)

{

T x1 = x;

T x2 = y;

}

T maximo(T a,T b)

{

return a > b ? a : b;

}

};

template<>

class max<bool>

{

bool x1;

bool x2;

max(bool x, bool y)

{

bool x1 = x;

bool x2 = y;

}

public:

bool maximo(bool a, bool b)

{

return a || b;

}

};

Aquí se puede apreciar como para el tipo bool la comparación se especializa, aplicando esta de una forma mas pertinente. En una especialización cuando se utiliza los tipos específicos se utiliza la definición de la especialización en vez de la del **Template** original. Una especialización posee el mismo nombre pero puede diferir en varias cosas del original (funciones, propiedades,...)

Cuando se quiere especializar algun subconjunto de los parametros de la entrada se utiliza lo que se conoce como especializacion parcial de **Template.**

**Singleton**

Singleton es un patrón de diseño creacional que asegura que una clase tenga una sola instancia, proveyendo un punto de acceso global a dicha instancia.

En el diseño singleton no se puede utilizar un constructor público, pues este siempre retorna un objeto debido a su diseño. Luego el constructor tiene que ser privado para poder asegurarnos que al constructor de la clase se accede una sola vez.

Hay diferentes maneras de diseñar un patrón singleton, no obstante todas tienen en común los siguientes pasos:

* Hacer el constructor por defecto privado.
* Crear un método estático que actúe como constructor. Cuando se llama a este método por primera vez llama al constructor privado y crea al objeto y lo guarda en un campo estático (dentro de la clase). Luego los próximos llamados a este método devuelven la dirección de memoria del objeto guardado.

Luego cada vez que se llame a la clase, será retornado siempre el mismo objeto.

Ventaja:

* Es seguro que una clase tiene una sola instancia de una clase

Desventajas:

* Viola el principio de la responsabilidad única.
* En ambientes con multi-thread necesita ser tratado con cuidado para evitar que varias thread creen un objeto singleton varias veces