7.7.—Compendio de ingeniería inversa para C++

Las clases de C++ son extensiones de structs C orientadas al objeto, por lo tanto es lógico intentar comprender las estructuras de datos repasando las características de la compilación de código C++. El C++ es demasiado complejo para detallar su comprensión con cuatro explicaciones. Por lo tanto lo que intentaremos será estudiar los rasgos sobresalientes y algunas diferencias entre la compilación con Microsoft Visual C++ y el GNU g++.

Un punto importante a tener en cuenta, es que hay que tener una comprensión buena del lenguaje C++ para entender bien la compilación de C++. Los conceptos de orientado al objeto, la herencia y el polimorfismo dificultan en gran medida aprender bien el código fuente. Intentar sumergirnos en estos conceptos a nivel de código ensamblado sin tener una comprensión a nivel de código fuente nos puede provocar un fuerte sentimiento de impotencia. En fin vamos a ello.

7.7.1.—El puntero this

El puntero **this** es un puntero disponible para todas las funciones C++ con elementos **nonstatic**. Siempre que una función es llamada el puntero **this** es inicializado apuntando al objeto utilizado para invocar a la función. Consideremos las siguientes llamadas a las funciones:

```
//objeto1, objeto2, y *tora_obj son todos del mismo tipo.
objeto1.elemento_func ( );
objeto2.elemento_func ( );
tora_obj->elemento_func ( );
```

En las tres llamadas a **elemento_func**, el puntero **this** carga los valores **&objeto1, &objeto2 y tora_obj**, respectivamente. Es fácil ver que **this** es pasado como el primer parámetro oculto en todas las funciones con elementos **nonstatic**. Tal como explicamos en la parte 4, Microsoft Visual C++ utiliza el acuerdo de llamada **thiscall** por lo cual pasa a **this** al registro **ECX**. El compilador GNU g++ maneja a **this** de la misma forma que si fuera el primer parámetro, el de más a la izquierda, a funciones con elementos **nonstatic** y empuja a la pila como último elemento, la dirección del objeto utilizado para invocar a la función antes de llamar a la función.

Desde el punto de vista de ingeniería inversa, el movimiento de una dirección al registro **ECX** antes de una llamada a una función, probablemente nos indica dos cosas. Primero, que el archivo ha sido compilado utilizando **Visual C++**. Segundo la función es un elemento de otra función. Cuando la misma dirección es pasada a dos o más funciones, podemos afirmar que todas estas funciones tienen la misma jerarquía de clase.

Dentro de una función, el uso de **ECX** antes de inicializarlo implica que el llamador debe haber inicializado a éste y es una posible indicación de que dicha función es un elemento de otra función, o que simplemente la función utiliza el acuerdo de llamada **fastcall**. Además, cuando se observa que una función elemento de otra pasa el puntero **this** a otras funciones, puede afirmarse que esas funciones son también elementos de la misma clase.

Para el código compilado utilizando **g++**, saber si las llamadas son realizadas a funciones elementos de otra es sencillo. Ya que, cualquier función que no tenga un puntero como su primer argumento, puede excluirse de que sea una función elemento de otra.

7.7.2.—Funciones virtuales y vtable

Las funciones virtuales en los programas C++, son las que proporcionan el medio para el comportamiento polimorfo. Para cada clase, o subclase por herencia, que contengan funciones virtuales, el compilador genera una tabla la cual contiene los punteros a cada función virtual de la clase. Dichas tablas reciben el nombre de **vtable**. Además, a cada clase que contenga funciones virtuales se le asigna un elemento dato adicional cuyo propósito es apuntar a la **vtable** apropiada en su ejecución. Dicho elemento se le da normalmente el nombre de **vtable pointer** y está distribuido como el primer elemento dato dentro de la clase. Cuando se crea un objeto en tiempo de ejecución, el **vtable pointer** es habilitado para que apunte a la **vtable** apropiada. Cuando ese objeto invoca a una función virtual, se selecciona la función correcta ejecutándose una mirada a la **vtable** del objeto. De esta forma, las **vtable** son el mecanismo fundamental en tiempo de ejecución para resolver las llamadas a las funciones virtuales. Unos cuantos ejemplos pueden ayudar para aclarar la utilización de las **vtable**. Consideremos las siguientes definiciones de clase C++:

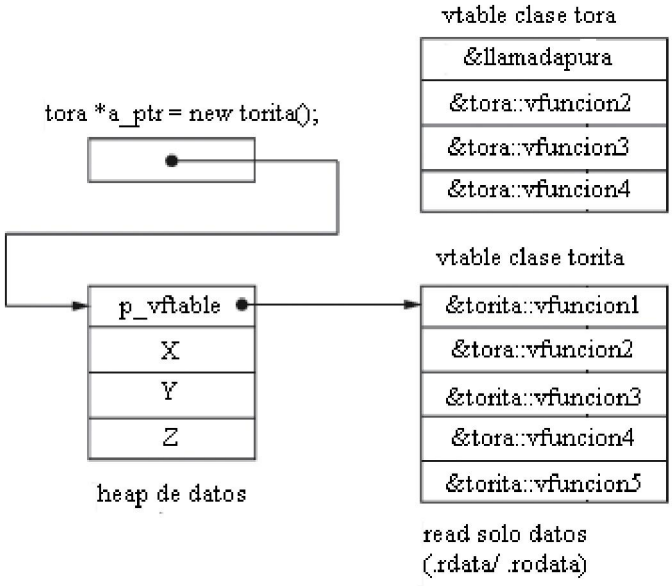
```
class tora
        {
        public:
          tora();
          virtual void vfuncion1() = 0;
          virtual void vfuncion2();
          virtual void vfuncion3();
          virtual void vfuncion4();
        private:
          int x;
          int y;
        };

class torita : public tora
        {
        public:
          torita();
          virtual void vfuncion1();
          virtual void vfuncion3();
          virtual void vfuncion5();
        private:
          int z;
        };
```

En este caso la clase **torita** es una subclase de **tora**. La clase **tora** contiene cuatro funciones virtuales, mientras que la clase **torita** contiene cinco, cuatro de **tora** más una nueva **vfuncion5**. Dentro de la clase **tora**, **vfuncion1** es una **función virtual pura**, ya que en su declaración se le asigna el valor de **0**. Las funciones virtuales puras no tienen ninguna ejecución en la clase donde se han declarado y deberá ser sobrescrita por una

subclase antes de que sea considerada concretamente por su clase. En otras palabras, no existirá ninguna función llamada **tora: :vfuncion1**, hasta que una subclase proporcione una ejecución de ella, ningún objeto podrá ser procesado. Como la clase **torita** proporciona dicha ejecución, podrán ser creados objetos de clase **torita**.

A primera vista la clase **tora** parece contener dos elementos de datos y la clase **torita** tres elementos de datos. Sin embargo la llamada a cualquier clase que contenga funciones virtuales, explícitamente o por herencia, también contiene un **vtable pointer**. Como consecuencia, los objetos procesados del tipo **tora** tienen en realidad tres elementos de datos, mientras que los objetos procesados del tipo **torita** tienen cuatro elementos de datos. En cada caso, el primer elemento dato es el **vtable pointer**. Dentro de la clase **torita**, el **vtable pointer** es heredado de la clase **tora** aunque se introduzca específicamente para la clase **torita**. En la figura siguiente se muestra un esquema de memoria simplificado en el cual un objeto tipo **torita** se ha distribuido dinámicamente. Durante la creación del objeto, el compilador se asegura que el **vtable pointer** del nuevo objeto apunte a la **vtable** correcta, **clase torita** en este caso.



Observemos que la **vtable** de la clase **torita** contiene dos punteros a funciones pertenecientes a la clase tora, **tora: :vfuncion2** y **tora: :vfuncion4**. Esto se debe a que la clase **torita** no hace caso de estas funciones porque las hereda de **tora**. También se muestra cómo se maneja una entrada de una función virtual pura. Como que no existe ninguna ejecución para la función virtual pura **tora: :vfuncion1**, no existe ninguna dirección disponible a guardar en el espacio de la **vtable** de **tora** respecto a **vfuncion1**. En estos casos, el compilador introduce la dirección de error de manipulación de función, a menudo nos muestra **purecall**, que en teoría nunca debería realizarse la llamada pero que si ocurriera por cualquier cosa, aborta el programa.

Una de las consecuencias de la presencia de un **vtable pointer**, es que IDA deberá explicarlo cuando manipule la clase. Como ya hemos dicho las llamadas de clases C++ son extensiones de estructuras C. Por lo tanto, podemos hacer que IDA utilice las características de definición de estructuras para definir el esquema de las clases C++. En el caso de las clases que contengan funciones virtuales, deberemos acordarnos de incluir un **vtable pointer** como primer campo dentro de la clase. Los punteros a **vtable** deben contabilizarse también en el tamaño total de un objeto. Esto aparentemente debe de

tenerse en cuenta en la distribución dinámica de un objeto cuando se utilice el operador **new**. Este operador se utiliza en C++ para la distribución dinámica de memoria en la mayoría de los casos en C se utiliza **malloc** sin embargo en el lenguaje C++ se utiliza **new** ya que **malloc** es meramente una función de librería. Por lo tanto el valor del tamaño pasado a **new** incluye el espacio utilizado por todos los campos declarados explícitamente en la clase, y cualquier superclase, así como cualquier espacio requerido para un **vtable pointer**.

En el siguiente ejemplo un objeto de clase **torita** se crea dinámicamente, y la dirección es guardada en un puntero a la clase **tora**. Luego el puntero es pasado a la función (**llamada_vfuncion**), la cual utiliza el puntero para llamar a **vfuncion3.**

```
void llamada_vfuncion (tora *a)
        {
           a->vfuncion3 ( );
        }

int main ( )
        {
           tora *a_ptr = new (torita);
           llamada_vfuncion (a_ptr);
        }
```

Debido a que **vfuncion3** es una función virtual, el compilador debe asegurarse en este caso, que **torita: :vfuncion3** es llamada ya que el puntero apunta a un objeto **torita**. En el siguiente desensamblado de la función **llamada_vfuncion** se muestra como se resuelve la llamada a la función virtual:

```
.text:004010A0
.text:004010A0 ; Attributes: bp-based frame
.text:004010A0
.text:004010A0 llamada_vfuncion proc near              ; CODE XREF: _main+37↓p
.text:004010A0
.text:004010A0 a               = dword ptr  8
.text:004010A0
.text:004010A0                 push    ebp
.text:004010A1                 mov     ebp, esp
.text:004010A3                 mov     eax, [ebp+a]
.text:004010A6                 mov     edx, [eax]
.text:004010A8                 mov     ecx, [ebp+a]
.text:004010AB                 mov     eax, [edx+8]
.text:004010AE                 call    eax
.text:004010B0                 pop     ebp
.text:004010B1                 retn
.text:004010B1 llamada_vfuncion endp
.text:004010B1
```

Como podemos observar el **vtable pointer** es leído desde la estructura en, línea abajo,

```
.text:004010A6                 mov     edx, [eax]
```

y guardado en el registro **EDX**. Luego como que el parámetro **a** apunta a un objeto **torita**, éste será la dirección de la **vtable torita**. Sigamos, línea abajo, la **vtable** es

```
.text:004010AB                 mov     eax, [edx+8]
```

Indexada para que lea el tercer puntero, en este caso la dirección de **torita: :vfuncion3**, y lo pase al registro **EAX**. Finalmente, línea abajo, la función virtual es llamada.

```
.text:004010AE                    call    eax
```

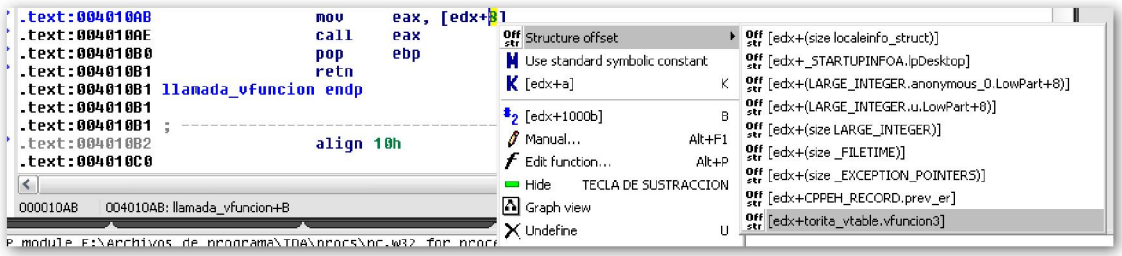
Observemos que la operación de indexación de la **vtable**, línea abajo, es muy parecida a

```
.text:004010AB                    mov     eax, [edx+8]
```

una operación referenciada a la estructura. De hecho, no difiere en nada de ella, por lo tanto es posible definir una estructura, como ya hemos aprendido anteriormente, la cual represente el esquema de la **vtable** de una clase y por consiguiente utilizar dicha estructura definida para hacer más comprensible el desensamblado, dicha estructura sería la siguiente:

```
00000000 torita_vtable   struc ; (sizeof=0x14)
00000000 vfuncion1       dd ?
00000004 vfuncion2       dd ?
00000008 vfuncion3       dd ?
0000000C vfuncion4       dd ?
00000010 vfuncion5       dd ?
00000014 torita_vtable   ends
```

Esta estructura definida nos permite formatear la operación referenciada a la **vtable**, de la siguiente forma:




```
.text:004010AB                    mov     eax, [edx+torita_vtable.vfuncion3]
```

7.7.3.—El ciclo vital del objeto

La comprensión del mecanismo por el cual los objetos son creados y destruidos, puede ayudarnos a revelarnos la jerarquía del objeto y sus relaciones con objetos anidados así cómo identificar rápidamente las funciones de la clase constructora y destructora. La función de una clase constructora es una función de inicialización la cual es llamada automáticamente cuando un objeto es creado. Una función destructora es opcional y será llamada cuando un objeto esté mucho tiempo sin utilizarse o similares.

Para distribuir objetos globalmente o de forma estática, se llaman a los constructores durante el inicio del programa y antes de entrar a la función **main**. Para los objetos distribuidos en **stack**, los constructores son llamados en el momento en que el objeto está al alcance de la función en donde se declara. En muchos casos, esto se realiza en la misma iniciación de la función en la que se declara. Sin embargo, cuando un objeto se declara dentro de un bloque de declaración, su constructor no es invocado hasta que dicho bloque haya sido declarado, si de algún modo se declara. Cuando un objeto es distribuido dinámicamente en el **heap** del programa, su creación se compone de dos pasos. En el primer paso se invoca al operador **new** para distribuir el objeto a memoria. En el segundo paso, se llama al constructor para inicializar el objeto. La diferencia entre

Microsoft Visual C++ y GNU g++ es que, Visual C++ se asegura de que el resultado de **new** no sea **null** antes de invocar al constructor.

Cuando se ejecuta un constructor, se realiza la siguiente secuencia de acciones:

1. Si la clase tiene una superclase, se llama al constructor de la superclase.

2. Si la calase tiene funciones virtuales, el **vtable pointer** es inicializado para que apunte a la **vtable** de la clase. Tengamos en cuenta, que esto puede sobrescribir un puntero a la **vtable** que había sido inicializado por la superclase, lo cual es lo que pretendíamos.

3. Si la clase contiene algún elemento dato que en sí mismo sea un objeto, entonces es invocado el constructor de cada elemento dato.

4. Finalmente se ejecuta el código específico del constructor. Este es el código en C++ que indica el comportamiento al constructor y es el especificado por el programador.

Los constructores no especifican ningún retorno tipo; sin embargo los constructores generados por Microsoft Visual C++ en realidad retornan **this** al registro **EAX.** Sin excepción, este un detalle de ejecución Visual C++, el cual permite a los programadores en C++ acceder al valor retornado.

Por otro lado, los destructores son llamados en orden inverso. Para objetos globales y estáticos, los destructores son llamados para que limpien el código ejecutado una vez se ha finalizado la función **main**. Los destructores para objetos distribuidos en **stack** son invocados de la misma forma que para objetos fuera de alcance. Los destructores para objetos distribuidos en el **heap** son invocados a través del operador **delete**, antes de que se realice la distribución de memoria para el objeto.

Las acciones ejecutadas por los destructores imitan a las ejecutadas por los constructores, con la excepción de que éstas se ejecutan en orden inverso. Estas son:

1. Si la clase tiene funciones virtuales, el **vtable pointer** del objeto es restaurado para que apunte a la **vtable** asociada a la clase. Este es requerido en el caso de que una subclase haya sobrescrito el **vtable pointer** como parte de su proceso de creación.

2. Se ejecuta el código del destructor, especificado por el programador.

3. Si la clase tiene elementos dato que sean en sí mismos objetos, se ejecuta el destructor de cada uno de los elementos dato.

4. Finalmente, si el objeto tiene una superclase, es llamado el destructor de la superclase.

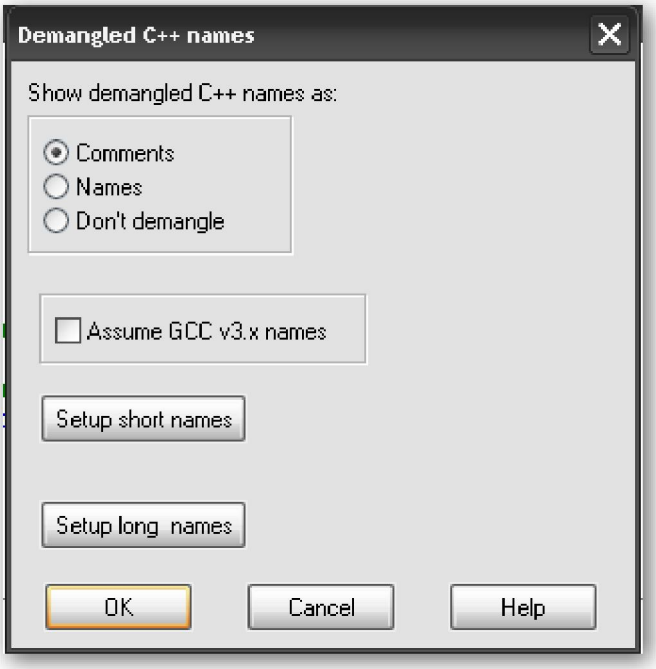
A través de la comprensión de cuando son llamados los constructores y destructores de una superclase, es posible trazar la jerarquía de herencia de un objeto siguiendo la cadena de llamadas a las funciones relacionadas con dicha superclase. Para finalizar con

las **vtable** veremos cómo son referenciadas dentro de los programas. Solamente existen dos circunstancias en las que se referencia directamente a una **vtable**, dentro del constructor y destructor de clase. Cuando localizamos una **vtable**, podemos utilizar las capacidades de referencias cruzadas en IDA, lo cual veremos en la siguiente parte, para rápidamente localizar todos los constructores y destructores asociados a la clase.

7.7.4.—Deformación de nombres (names)

En inglés, **name decorated o name mangling**, es el mecanismo que los compiladores C++ utilizan para distinguir la gran cantidad de distintas versiones cargadas (**overload**) de una función. En C++, la sobrecarga de función permite a los programadores utilizar el mismo nombre para varias funciones. El único requisito es que cada versión de la función sobrecargada debe diferir, de las otras versiones, en la secuencia y/o la cantidad de tipos de parámetro que reciba la función. En otras palabras, cada función prototipo debe ser única. Por lo tanto para generar nombres únicos para las funciones sobrecargadas, los compiladores añaden al nombre de la función caracteres adicionales respecto a distintas partes de información de la función. La información adjuntada normalmente describe el retorno tipo de la función, la clase a la cual pertenece la función y la secuencia de parámetros (tipo y orden) requeridos al llamar a la función.

La deformación de nombres es una ejecución particular del compilador para programas C++, con lo cual no es parte del lenguaje C++. Por otro lado los distribuidores de compiladores han desarrollado cada cual su propia deformación, a menudo incompatibles entre ellas. Por fortuna, IDA entiende las convenciones de deformación empleadas por Microsoft Visual C++ y GNU g++, así como las de otros compiladores. Por defecto cuando IDA encuentra un nombre deformado dentro de un programa, IDA muestra la deformación como un comentario en cualquier parte del desensamblado en donde aparezca. Las opciones de IDA para deformar un nombre se pueden seleccionar en el diálogo mostrado a continuación, realizando la acción **Options > Demangled Names.**



Las tres opciones principales que controlan la deformación de nombres son las mostradas como **Comments**; los nombres son mostrados como comentarios, **Names**;

los nombres son deformados y **Don't demangle**; no se ejecuta deformación. Mostrar los nombres deformados como **Comment** nos resultaría una vista similar a la siguiente:

Y una llamada a la función se vería:

Sin embargo con la opción, mostrar **Names** deformados la función sería lo siguiente:

Y su llamada sería:

La selección de **Assume GCC v3.x names**, se utiliza para distinguir entre el esquema de deformación utilizado en g++ versión 2.9.x y la usada en la versión 3.x y posteriores. En circunstancias normales, IDA detectará automáticamente las convenciones de deformación utilizadas en el código compilado por g++. Los botones **Setup short names** y **Setup long names**, nos proporcionan un control sobre el formato de los nombres deformados con una buena cantidad de opciones documentadas en la Ayuda de IDA.

Ya que los nombres deformados nos transmiten tanta información con respecto a la firma de cada función, estos nos reducen el tiempo necesario para comprender el número y tipos de parámetros pasados a una función. Cuando dentro de un binario disponemos de los nombres deformados, la característica de deformación de IDA nos muestra instantáneamente los parámetros tipo y el retorno tipo de todas las funciones que tengan su nombre deformado. En cambio, para cada función que no utilice el nombre deformado, el tiempo utilizado para su análisis de flujo de datos dentro y fuera de la función para determinar su firma, será mucho mayor.

7.7.5.—Identificación de tipos en tiempo de ejecución

C++ nos proporciona operadores que nos permiten en tiempo de ejecución, determinar **typeid** y verificar **dynamic_cast** el tipo de dato de un objeto detectado. Para facilitar estas operaciones los compiladores C++ deben adjuntar la información de tipo dentro del programa binario y ejecutar procedimientos por medio de los cuales pueda determinarse el tipo de un objeto polimorfo con certeza, a pesar del tipo del puntero que pueda ser referenciado para acceder al objeto. Desafortunadamente, para un nombre deformado, la ejecución con detalle del **Runtime Type Identification (RTTI)** del compilador tendrá un problema de lenguaje, ya que no existen estándares por los cuales los compiladores pueden poner en práctica sus capacidades **RTTI**.

Daremos una breve revisión a las similitudes y diferencias entre las ejecuciones **RTTI de Microsoft Visual C++ y GNU g++**. Específicamente los únicos detalles que veremos serán los concernientes a la localización de la información **RTTI**, y una vez localizada cómo conocer el nombre de la clase a la que pertenece dicha información.

Consideremos el siguiente programa el cual utiliza polimorfismo:

```
class clase_abstracta
        {
           public:
                virtual int vfuncion ( ) = 0;
        };

class clase_concreta : public clase_abstracta
        {
           public:
                clase_concreta ( );
                int vfuncion ( );
        };

void imprime_tipo (clase_abstracta *p)
        {
           cout << typeid (*p).name ( ) << endl;
        }

int main ( )
        {
           clase_abstracta *sc = new clase_concreta ( );
           imprime_tipo (sc);
        }
```

La función **imprime_tipo** debe imprimir correctamente el tipo del objeto existente apuntado por el puntero **p**. En este caso es trivial comprender que **clase_concreta** debe ser impresa basándonos en el hecho de que un objeto de **clase_concreta** es creado en la función **main**. La pregunta que debemos contestarnos aquí es la siguiente: ¿Cómo hacer que la función **imprime_tipo**, y más específicamente el operando **typeid**, sepa a que tipo conocido del objeto **p** debe apuntar?

La respuesta es sorprendentemente simple. Debido a que cada objeto polimorfo contiene un puntero a una **vtable**, los compiladores intentan establecer de hecho la localización por ambas partes, clase y vtable de la clase, el tipo de información. Específicamente, el compilador coloca un puntero justo antes que el de la vtable. Este puntero apunta a una estructura la cual contiene la información utilizada para determinar el nombre de la clase que posee dicha vtable. En el código de g++, este puntero apunta a una estructura **type_info**, la cual contiene un puntero al nombre de la clase. En Visual C++, el puntero apunta a una estructura Microsoft **RTTICompleteObjectLocator**, que a su vez contiene un puntero a una estructura **TypeDescriptor**. Dicha estructura contiene un conjunto de caracteres que especifica el nombre de la clase polimórfa.

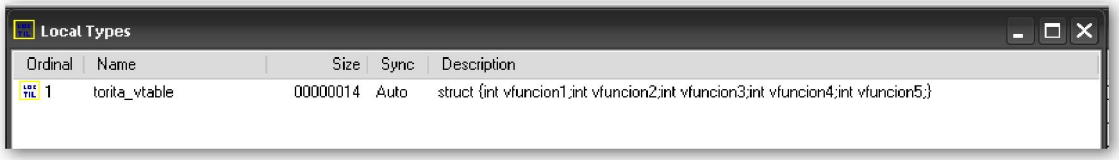
Es importante comprender que la información RTTI sólo es requerida en programas C++ utilizando el operador **typeid** o **dynamic_cast**. La mayoría de los compiladores proporcionan opciones para deshabilitar la generación de RTTI en los binarios que no se requiera; por lo tanto no debe sorprendernos si en alguna ocasión no encontramos la información RTTI.

7.7.6.—Relaciones de herencia

Si profundizamos en ciertas ejecuciones RTTI, nos encontraremos con que es posible desenmarañar las relaciones de herencia, sin embargo es necesario comprender la ejecución particular RTTI del compilador. También, hay que tener en cuenta que no existirá información RTTI si el programa no utiliza los operadores **typeid** o **dynamic_cast** ¿Qué técnicas podemos emplear para determinar las relaciones de herencia entre las clases C++ cuando no dispongamos de información RTTI?

El método más simple para determinar una jerarquía de herencia es observar la cadena de llamadas realizadas a los constructores de la superclase cuando se crea un objeto. El problema más grande de esta técnica es la utilización de constructores **inline**. En los programas C/C++ una función declarada como **inline** es tratada por el compilador como una macro, y el código de la función será expandido en lugar de realizarse una llamada explícita a la función. En el momento en que exista una declaración de llamada en lenguaje ensamblador se toma como una falsa llamada a la función que se está llamando, la utilización de las funciones **inline** tiende a ocultar el hecho de que una función se está utilizando. La utilización de dichos constructores **inline**, hace imposible comprender que en realidad se ha llamado a un constructor de la superclase.

Una alternativa significativa para determinar las relaciones de herencia es realizar la comparación y análisis de las **vtables**. Por ejemplo si comparamos la vtable siguiente,



observamos que la **vtable torita** contiene dos punteros iguales que los que aparecen en la **vtable tora**. De ahí podemos deducir que la clase tora y la clase torita están relacionadas de alguna forma. ¿Pero y si torita es un subclase de tora o tora es una subclase de torita, cómo saberlo? En estos casos podemos aplicar las siguientes normas cada una por sí misma o combinadas para poder entender la naturaleza de su relación.

** Cuando dos vtable contienen el mismo número de entradas, las dos clases correspondientes a ellas pueden tener alguna relación de herencia.

** Cuando una vtable de una clase X contiene más entradas que la vtable de la clase Y, la clase X puede ser una subclase de Y.

** Cuando una vtable de una clase X contiene entradas que también se encuentran en la vtable de una clase Y, entonces existirá una de las siguientes relaciones: X es una subclase de Y, Y es una subclase de X o X y Y son ambas subclases de una superclase Z común a las dos.

** Cuando la vtable de una clase X contiene entradas las cuales también existen en la vtable de la clase Y y la vtable de la clase X contiene al menos una entrada de llamada pura la cual no está presente en la vtable de Y, entonces la clase Y es una subclase de la clase X.

Aunque en el listado anterior de ningún modo se ha incluido todas las normas, podemos utilizar dichas orientaciones para deducir la relación entre las clases ejemplo tora y torita. En este caso, se confirman las tres últimas normas, pero la última es la más concluyente, la cual basándonos en el análisis de la vtable podemos afirmar que la clase torita es una subclase de la clase tora.

Performance Bigundill@