Manual del Lenguaje
Meta D++

Autora: María Belén Lícari

CONTENIDO

ESTRUCTURA DEL MANUAL

El eje central de este manual es el lenguaje de programación de Meta D++. Los temas incluidos en el manual se organizan por capítulos, los cuales se mencionan a continuación:

Primera Parte:

- ✓ **Capítulo I-Introducción:** Antes de empezar a describir el manual es necesario explicar conceptos básicos de la tecnología de desarrollo de software LayerD, cuáles son sus características, como también explicar algunos conceptos del lenguaje de programación Meta D++, como se usan los compiladores, etc.
- ✓ **Capítulo II-Tipos de Datos y Variables:** En este capítulo se describirán los tipos de datos que se pueden manejar en Meta D++.
- ✓ **Capítulo III-Operadores:** Se describirán los operadores que se pueden manejar en el lenguaje de programación de Meta D++
- ✓ **Capítulo IV-Instrucciones:** Se explicarán los tipos de instrucciones soportadas por el lenguaje de programación Meta D++
- ✓ **Capítulo V-Conceptos de Orientación a objetos:** En este capítulo se describirán conceptos relacionados con el manejo de Clases, métodos, miembros de clases,etc
- ✓ **Capítulo VI-Manejo de errores:** Se describirá en este capítulo como se manejan los errores que puedan ocurrir en el momento de la ejecución de una aplicación y formas de evitar que los mismos ocurran.

Segunda Parte:

- ✓ **Tiempo de Compilación vs. Tiempo de Ejecución:** se proporciona una introducción teórica a los principales conceptos necesarios para programar generación de código, realizar introspección del programa que se está compilando, generar estructuras semánticas, etc.
- ✓ **Guía paso a paso de Programación en Tiempo de Compilación:** es una rápida guía paso a paso donde se explica cómo programar extensiones de tiempo de compilación y utilizarlas desde programas cliente. Si bien no cubre todos los temas posibles incluye una buena cantidad de ejemplos sobre como tomar argumentos desde los programas clientes y generar código en tiempo de compilación.

Primera Parte

Conceptos Básicos y Orientación a Objetos tradicional

1 INTRODUCCIÓN

1.1 Similitud con Otros Lenguajes de Programación

El lenguaje de programación Meta D++ posee una sintaxis y estructura léxica similar a otros lenguajes de programación como C++, Java y C#. Semánticamente el lenguaje Meta D++ duplica las características de lenguaje intermedio de LayerD, el lenguaje Zoe.

1.2 Introducción al framework LayerD

LayerD es una nueva tecnología de desarrollo de software – al menos a la hora de escribir éste manual. Es de libre acceso, tanto la documentación como los compiladores y herramientas. LayerD ha surgido originalmente como una idea de un lenguaje estándar que pudiera traducirse con medios automáticos a diferentes lenguajes de programación de alto nivel, ello fue allá por el año de 1999. Luego la idea fue abandonada temporalmente por falta de tiempo y recursos para completarla, hasta que a mediados de 2002 se retomo y reinterpretó la idea original (incorporando el objetivo de desarrollo de software multiplataforma con "*Costo Cero de Ejecución*") hasta convertirla en lo que hoy es LayerD, pero nuevamente la escasez de recursos y tiempo paralizo el avance en la concreción práctica de la tecnología.
LayerD se caracteriza por:

- Posibilidad de generación de software multiplataforma real sin incurrir en perdida de performance. Por ejemplo, se pueden construir programas que compilen sin cambios para .NET, Java y otras plataformas que se encuentren disponibles.
- Programación Orientada a Objetos. Incluyendo programación orientada a objetos en tiempo de compilación.
- Creación de nuevas estructuras semánticas. Usando el tiempo de compilación y los tipos especiales accesibles se pueden implementar nuevas estructuras semánticas al los lenguajes de alto nivel sin depender de los diseñadores del lenguaje o los fabricantes de los compiladores.
- Programación de herramientas RAD incorporadas nativamente. Se pueden programar herramientas RAD que corran dentro del compilador y sean aplicables a más de un lenguaje de alto nivel. Al correr las utilidades dentro del compilador intermedio Zoe son independiente del entorno de programación utilizado.
- Posibilidad de Extensión dinámica del compilador. El compilador intermedio Zoe, al ser programable su tiempo de compilación y poseer reflexión completa del código que se está compilando, permite programar módulos de extensión para agregar nueva funcionalidad al compilador desde nuevas estructuras semánticas hasta la incorporación de lenguajes de dominio especifico o la generación masiva de código. Todo usando orientación a objetos tradicional y permitiendo programar de forma modular.

LayerD puede entenderse en base a los siguientes cuatro componentes:

1. **Meta_Lenguajes:** es lo único que todo programador LayerD debe conocer, ya que un Meta-Lenguaje no es ni más ni menos que el lenguaje de alto nivel en el cual los programadores escribimos los programas. Un Meta-Lenguaje no se diferencia en nada de un lenguaje de alto nivel actual como Java, C#, VisualBasic, Eiffel, Ada, Smalltalk, Perl, etc., de hecho los Meta-Lenguajes no son en realidad una tecnología nueva, simplemente son lenguajes de programación que se adaptan al modelo de desarrollo de LayerD, es decir que se adaptan al estándar LayerD. Actualmente los desarrolladores de

la tecnología LayerD tienen en sus planes el lanzamiento de tres Meta-Lenguajes en una primera etapa, estos son: Meta D++, LayerD-Basic y Argentino. Meta D++ es un lenguaje orientado a objetos el cual es capaz de aprovechar todas las capacidades de LayerD, con una sintaxis similar a C++. LayerD-Basic es otro lenguaje orientado a objetos destinado a programadores de alto nivel con una sintaxis estilo Basic. Argentino es un lenguaje orientado a objetos enfocado en programadores de alto nivel (al igual que LayerD-Basic) pero con sintaxis en lengua castellana. Si bien los Meta-Lenguajes que próximamente estarán disponibles serán imperativos y orientados a objetos, no es un requisito que los Meta-Lenguajes lo sean, de hecho se espera que otras organizaciones implementen lenguajes orientados a paradigmas diferentes como lenguajes funcionales y lenguajes lógicos. El requerimiento que diferencia a un lenguaje LayerD de uno clásico es el resultado de su compilación, mientras que el compilador de un lenguaje clásico genera código objeto, pseudo código o una especie de bytecode, los compiladores de los Meta-Lenguajes generan código Zoe, es decir que la salida de un compilador LayerD es el mismo programa pero traducido a un lenguaje estándar y central a la tecnología, luego se verá la importancia de dicho requerimiento.

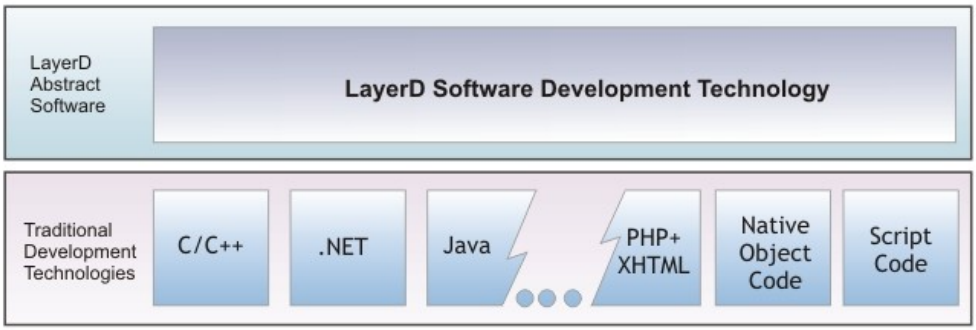
2. **Lenguaje Zoe:** es la piedra angular de la tecnología LayerD, Zoe no es más que un lenguaje de alto nivel imperativo estandarizado por la tecnología, su función es similar al bytecode en la tecnología Java, sin embargo Zoe no se genera en forma de código binario para una maquina virtual, sino que Zoe se escribe en un dialecto XML, el dialecto Zoe, es decir que Zoe se puede entender a simple vista, es fácil y seguro de transferir a cualquier plataforma y permite que en el mismo archivo se incluyan datos relacionados con el código utilizando otros dialectos XML. Como todos los Meta-Lenguajes están obligados a generar código Zoe, luego se lo utiliza para incluir en él una serie de características básicas que se vuelven común a todos los Meta-Lenguajes, esa característica es la posibilidad de orientación a objetos y lo más importante la capacidad para extender el lenguaje y el compilador utilizando "**classfactorys**". Más adelante se revelarán los secretos y fortalezas de Zoe, los cuales hacen posible la programación multiplataforma real, por ahora es importante que sepa que Zoe es estándar y que todos los Meta-Lenguajes deben compilarse a código Zoe.

3. **Compilador Zoe:** como todo lenguaje de programación Zoe requiere un compilador, sin embargo lo diferente del compilador Zoe, comparado con la mayoría de los lenguajes más utilizados actualmente, es su diseño modular y extensible, el compilador Zoe provee los mecanismos básicos de la compilación del lenguaje Zoe, sin embargo no provee ningún mecanismo estándar para generar el código final (a dicho proceso lo realizan los Generadores de Código Zoe), el compilador simplemente se limita a realizar el análisis semántico y la ejecución de las denominadas "**classfactorys**" del lenguaje Zoe, luego los generadores de código son los encargados de terminar el proceso. En la siguiente sección donde se explica el proceso de transformación desde código LayerD hasta tener el programa listo para el enlazado se explicará en mayor profundidad su objetivo y como lo logra.

4. **Generadores de Código Zoe:** como el compilador Zoe no genera el producto final necesario antes del ensamblado, los encargados de realizar dicho proceso, es decir completar la compilación de Zoe, son los módulos de salida. Mientras que en la mayoría

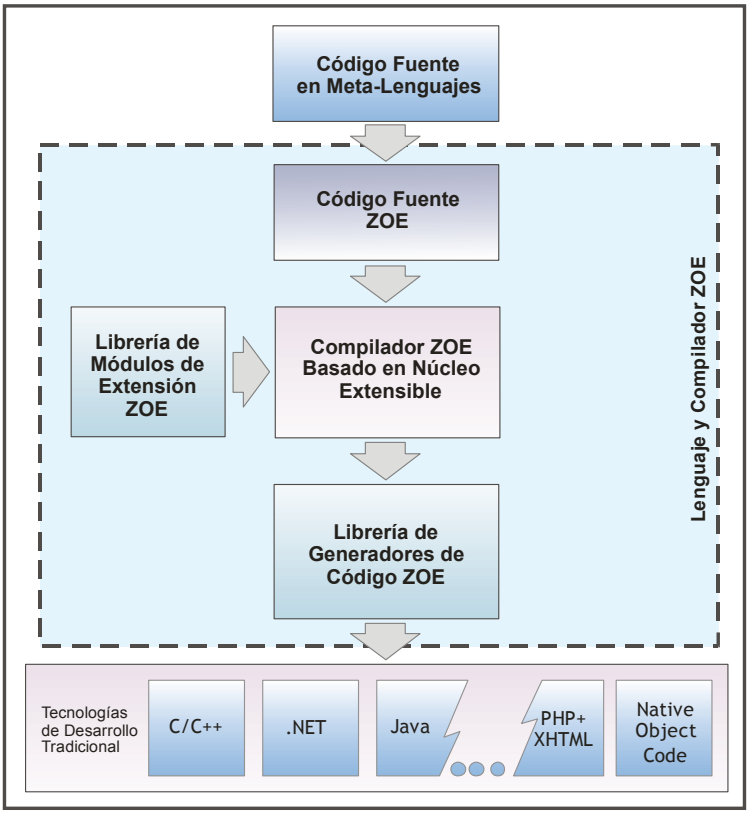
de los lenguajes los compiladores realizan todo el trabajo desde el análisis sintáctico hasta la generación de código, en LayerD, la generación de código se encuentra modularizada en los Módulos de Salida Zoe, es decir que en LayerD es posible generar software para tantas plataformas de destino como Generadores de Código Zoe se posean, además la interfaz entre el compilador Zoe y los generadores de código se encuentra estandarizada por la tecnología, por lo cual cualquier fabricante de software de base puede desarrollar y comercializar un generador de código, el cual funcionará con todos los compiladores Zoe. El objetivo es simple y fácil de entender, en vez de tener un generador de código fijo en el compilador (por ejemplo generador de código objeto para plataforma x86), los generadores de código permiten "enchufar" la generación de código para diferentes plataformas de destino a cualquier compilador Zoe; por ejemplo basta con disponer de un generador de código a lenguaje C, para poder compilar cualquier programa LayerD en cualquier plataforma que posea un compilador de C, o bien con un módulo de salida a bytecode, se puede compilar cualquier programa LayerD para correr en maquinas virtuales Java, como puede imaginar los generadores de código pueden generar código para cualquier plataforma de hardware o software

Al nivel más alto, LayerD es una "capa" construida sobre las tecnologías existentes, como muestra la siguiente imagen:



El software desarrollado con LayerD es potencialmente abstracto por poseer mecanismos prediseñados para ser construido en prácticamente cualquier tecnología de nivel inferior disponible actualmente sin perdida en rendimiento. Aún cuando no utilice LayerD para desarrollar software multiplataforma es posible beneficiarse de su arquitectura para obtener beneficios inéditos en la gran mayoría de los lenguajes “main-stream” en uso actualmente como es la capacidad de extensión de los Meta-Lenguajes y la reflexión programable en tiempo de compilación.

El proceso usado en LayerD para desarrollar software abstracto se muestra a continuación:



El proceso es el siguiente:

1. Se escribe el software en un Meta-Lenguaje, un "Meta-Lenguaje" es simplemente un lenguaje de alto nivel como C++, Java, C#, PHP, etc.
2. El código fuente LayerD es traducido a código Zoe el cual es un lenguaje estándar y central a la tecnología LayerD que se escribe en XML y esta destinado a ser procesado por compiladores.
3. El compilador modular basado en núcleo Zoe utiliza "Módulos de Extensión" para realizar la adaptación en el alto nivel al código fuente LayerD original para adaptar los protocolos e interfaces de la plataforma de implementación.
4. Finalmente un Modulo de Salida Zoe procesa la salida del compilador Zoe traduciendo una versión especial de código Zoe a un código apropiado para la plataforma de destino, ello puede ser código fuente en un lenguaje tradicional, bytecode o código nativo.

LayerD fue diseñado primariamente para desarrollar software abstracto e independiente de la plataforma, sin embargo el diseño modular basado en núcleo del compilador Zoe puede ser utilizado para beneficiarse con muchas capacidades inéditas en el común de las herramientas actuales de desarrollo de software, como ser:

- Es posible combinar programación Orientada a Objetos con programación Orientada a Aspectos sin la necesidad de herramientas externas o extensiones en los Meta-Lenguajes.
- Puede desarrollar sus propias Estructuras Semánticas, y hacerlo independientemente del Meta-Lenguaje. Los Meta-Lenguajes pueden ser extendidos por uno mismo agregando nuevas capacidades como lenguajes embebidos, estructuras semánticas, funciones sensitivas al contexto, y cualquier otra construcción que uno sea capaz de imaginar.

- Es posible agregar nuestros propios tests al proceso de compilación, por ejemplo para controlar si los usuarios de nuestros componentes utilizan el protocolo esperado para un programa cliente o no y emitir errores o advertencias dinámicamente en el proceso de compilación.
- Por lo que en LayerD usted puede extender los lenguajes de alto nivel sin la necesidad de depender de los diseñadores o implementadores originales del lenguaje y puede cambiar la forma de construir programas agregando nuevas palabras claves, construcciones semánticas o lenguajes embebidos que sean beneficiosos para su proyecto en particular, y expresar todo ello de una forma totalmente abstracta y orientada a objetos independiente de la plataforma y pudiendo utilizar toda la infraestructura existente sin perder una sola línea de código en la que haya invertido en el pasado.

1.3 El Lenguaje de Programación Meta D ++

Meta D++ es un lenguaje imperativo, orientado a objetos, adicionalmente soporta todas las características innovadoras de la tecnología LayerD como classfactorys y compilación interactiva.
El lenguaje Meta D++ ha sido desarrollado conjuntamente con Zoe, de hecho se puede decir que Meta D++ es una versión "para personas" de Zoe (o al revés que Zoe es una versión para compiladores y herramientas de Meta D++), por ello encontrará mucha de la especificación de Meta D++ idéntica a la de Zoe cuando se encuentre disponible, sobre todo en cuanto a la semántica del lenguaje.
La sintaxis de Meta D++ tiene mucho en común con C++, Java, C# y otros lenguajes orientados a objetos que siguen dicha sintaxis, la semántica de las construcciones más habituales también siguen lo ya establecido por otros lenguajes orientados a objetos ampliamente utilizados, a excepción de modificaciones necesarias por las características de LayerD de ser una tecnología de desarrollo multiplataforma y flexible. Las mayores innovaciones en Meta D++ las encontrará en las classfactorys y su dúctil sistema de tipos y administración de memoria (que de hecho duplica las características de Zoe).

Un Hola Mundo en Meta D++:

```
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
using DotNET::System;

namespace Test{
   public class App{
   public:
      static void Main(string^[] args){
         Console::WriteLine("Hola Mundo");
      }
   }
}
```

Otro Hola Mundo en Meta D++:

```
Zoe::ConsoleProgram::New
{
   Console::WriteLine("Hello world!!");
};
```

Como puede apreciarse, si se requiere utilizar un tipo importado desde .NET simplemente se lo utiliza de la misma forma en la que se lo haria desde cualquier otro lenguaje .NET teniendo en cuenta que los tipos importados se encuentran dentro del espacio de nombres “DotNET”. El segundo ejemplo de Hola Mundo muestra como si es necesario en Meta D++ los programas pueden poseer una estructura completamente declarativa y simplificada.

1.4 Comentarios

Los comentarios en Meta D++ siguen la convención del lenguaje C++.
Para los comentarios de línea empezados con una doble barra ‘*//*’ y los comentarios de bloque empezados con ‘*/**’ y terminados con ‘**/*’.

```
// Ejemplo:
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
using DotNET::System;

namespace Test{
   class A{
      // ESTO ES UN COMENTARIO CORTO
      /* ESTO ES UN COMENTARIO
      LARGO QUE OCUPA MAS DE UNA LINEA*/
      static void Main(string^[] args){
         Console::WriteLine("Esto es una prueba");
      }
   }
}
```

Los comentarios en Meta D++ se corresponden en el lenguaje Zoe con nodos de tipo XplDocumentation (nodos “documentation”) ó con el contenido del atributo “doc” de los nodos de un programa Zoe. Por esto, en Meta D++ los comentarios poseen una significación semántica y pueden ser analizados y transformados en tiempo de compilación y procesados por los generadores de código Zoe.
Por esta misma razón los comentarios en Meta D++ no pueden ir en cualquier parte del código. Sin embargo, los lugares naturales como se dentro del cuerpo de espacios de nombres, entre miembros de clase o entre instrucciones son aceptados; por lo tanto rara vez deberá acomodar los comentarios en un lugar específico, además esto ayudará a mantener la legibilidad del código.

1.5 Utilización de los compiladores

Para compilar un programa Meta D++ desde la línea de comandos debe realizar dos pasos:

- Compilar el fuente con el compilador de Meta D++ para obtener el código intermedio Zoe.
- Compilar el código intermedio con el compilador Zoe.

El compilador de Meta D++ es “metadppc.exe” y el compilador Zoe es “zoec.exe”. Por ejemplo, si posee el fuente “HelloWorld.dpp” se procede de la siguiente forma:

```
c:\layerd\bin>metadppc.exe HelloWorld.dpp
c:\layerd\bin>zoec.exe HelloWorld.zoe
```

La secuencia de comandos anterior producirá "`HelloWorld.exe`", note que el compilador de Meta D++ produce archivos de extensión `.zoe` por defecto. Para compilar un programa que posea más de un fuente primero compile los fuentes Meta D++ llamando al compilador `metaddc` y luego utilice el compilador `zoec` una vez, ejemplo:

```
c:\layerd\bin>metadppc.exe HelloWorld_Source1.dpp
c:\layerd\bin>metadppc.exe HelloWorld_Source2.dpp
c:\layerd\bin>zoec.exe HelloWorld_Source1.zoe HelloWorld_Source2.zoe -pn:HelloWorld
```

La opción "`-pn:HelloWorld`" indica que el nombre del programa Zoe el cual será utilizado como nombre de archivo de salida, por tanto se generará el archivo "`HelloWorld.exe`".

1.5.1 Compilar librerías dinámicas

Para compilar una librería dinámica en lugar de un ejecutable utilice la opción "-lib" del compilador Zoe:

```
c:\layerd\bin>metadppc.exe HelloWorldLib.dpp
c:\layerd\bin>zoec.exe HelloWorldLib.zoe -lib
```

1.5.2 Compilar y utilizar Classfactorys

Si bien es posible declara y utilizar classfactorys en un mismo programa e incluso en un mismo fuente, aquí se explicara cómo utilizar classfactorys escribiéndolas en un archivo y programa separado al programa cliente.
Asumiendo que se poseen los archivos "`MyClassfactory.dpp`" y "`MyClassfactoryClient.dpp`" con la classfactory y el programa cliente respectivamente, se debe proceder como sigue con los compiladores:

```
c:\layerd\bin>metadppc.exe MyClassfactory.dpp
c:\layerd\bin>zoec.exe MyClassfactory.zoe -ae
Extension Added: MyClassfactory

c:\layerd\bin>metadppc.exe MyClassfactoryClient.dpp
c:\layerd\bin>zoec.exe MyClassfactoryClient.zoe
```

Primero debe compilarse la classfactory y agregarla a la librería de classfactorys del compilador Zoe utilizando el modificador "`-ae`" (agregar extensión) y luego compilar el programa cliente. Como muestra el ejemplo el compilador Zoe debe mostrar un mensaje indicando que la extensión se agrego al compilador.

Para ver las extensiones instaladas en el compilador Zoe y desinstalar extensiones utilice respectivamente las opciones "`-le`" y "`-re:EXTENSION_NAME1,EXTENSION_NAME2,..`" (O las opciones equivalentes "`-listextensions`" y "`-removeextensions`"). Ejemplo:

```
c:\layerd\bin>zoec.exe -le
Compilador ZOE - Interfaz de Usuario de Consola.
Extensiones Instaladas: 12

Nombre de Tipo, Archivo del Modulo, Interactive, Active
-------------------------------------------------------
Zoe.Utils, .\.\FactoriesLib\ZoeUtilsTemplates1.dll, False, True
Zoe.iUtils, .\.\FactoriesLib\ZoeUtilsTemplates1.dll, True, True
Zoe.Logger, .\.\FactoriesLib\ZoeUtilsTemplates1.dll, False, True
DataSample.MyType, .\.\FactoriesLib\DataSampleTemplates1.dll, False, True
```

```
DataSample.GUI, .\.\FactoriesLib\DataSampleTemplates1.dll, False, True
DataSample.iASPNET, .\.\FactoriesLib\DataSampleTemplates1.dll, True, True
DataSample.ASPNET, .\.\FactoriesLib\DataSampleTemplates1.dll, False, True
DataSample.Concurrent, .\.\FactoriesLib\DataSampleTemplates1.dll, False, True
DataSample.ClassGenerator, .\.\FactoriesLib\DataSampleTemplates1.dll, False, True
DataSample.ProgramChecks, .\.\FactoriesLib\DataSampleTemplates1.dll, False, True
DataModel2.Model, .\.\FactoriesLib\DataModelTemplatesPablo1.dll, True, True
DataModel.Model, .\.\FactoriesLib\DataModelTemplates1.dll, False, True

c:\layerd\bin>zoec.exe –re:DataSampleTemplates
Compilador ZOE - Interfaz de Usuario de Consola.
Extension DataSampleTemplates eliminada. Classfactorys eliminadas: 7
```

Para ver más opciones del compilador Zoe y el compilador Meta D++ ejecute los compiladores sin proporcionar argumentos para mostrar la ayuda en la línea de comandos.

1.5.3 Compilar para otras plataformas

Por defecto el compilador Zoe construido para .NET genera módulos para dicha plataforma utilizando el Generador de Código Zoe para .NET.
Si se desea compilar un programa multiplataforma para otra plataforma que no sea la por defecto se debe utilizar la opción "`-p:PLATAFORMA`" del compilador Zoe, por ejemplo para compilar utilizando el Generador de Código Zoe para Java:

```
c:\layerd\bin>metadppc.exe HelloWorld.dpp
c:\layerd\bin>zoec.exe HelloWorld.zoe –p:java
```

Si no se posee un Generador de Código adecuado para la plataforma indicada o no existen las Classfactorys adecuadas que soporten dicha plataforma se informaran los errores.

2 TIPO DE DATOS Y VARIABLES

2.1 Definición y Tipos de Variables

Una variable representa un espacio de almacenamiento en memoria al que puede ser asignado un valor y luego éste leído. Toda variable en Meta D++ posee un tipo que determina los valores que pueden ser asignados a las mismas. Para poderla utilizar variables antes hay que definirla indicando cual será su nombre y cuál será el tipo de datos que podrá almacenar, lo que se hace siguiendo la siguiente sintaxis:

```
   <tipoVariable> <nombreVariable>;
```

```
//Ejemplo de  definición de variables
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
using DotNET::System;

namespace Test{
   class APP{
   public:
      static void Main(string^[] args){
         //declaración de la variable local a;
         int a;
         a = 0;
      }
   }
}
```

Recomendaciones para nombrar variables

Para nombrar variables es recomendable utilizar alguno de los clásicos enfoques:

- ✓ **Camel Casing:** Con esta forma de definición primera letra de cada palabra se escribe en mayúscula. Por ejemplo: **N**ombre**V**ariable
- ✓ **Pascal Casing:** Se declara en mayúscula la primera letra de la segunda palabra. Por ejemplo: **n**ombre**V**ariable

```
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   public class Persona{
   public:
      int edad;
      //definición de campo utilizando pascalCasing
      string^ nombrePersona;
      //definición de campo utilizando CamelCasing
      string^ GetNombrePersona(){
         return nombrePersona;
      }
      //constructor
      Persona(int edad){
         this.edad = edad;
         nombrePersona = "belen";
      }
   }
}
```

Variables

Las variables almacenan valor que puede cambiar cuando una aplicación se está ejecutando. Tiene seis elementos básicos, los cuales se describen a continuación

- ✓ **Nombre**: es la palabra que identifica a la variable en el código
- ✓ **Dirección**: es la ubicación en la memoria donde se almacena el valor
- ✓ **Valor**: valor de la dirección de la memoria
- ✓ **Vida**: es el intervalo de tiempo durante el cual la variable es válida
- ✓ **Ámbito**: es el conjunto de todo el código que puede acceder y utilizar la variable
- ✓ **Tipo de Dato**: es el tipo y tamaño de datos que la variable puede almacenar.

Tipos de Variables

El lenguaje Meta D++ define diversos tipos de variables, los cuales se definen a continuación.

Variables Locales

Una variable local es una variable declarada dentro un bloque por una instrucción de declaración, un bloque for, un bloque foreach o una cláusula catch de una instrucción try.
El tiempo de vida de la variable local se corresponde desde la instrucción en la que es declarada dentro del bloque hasta la finalización del bloque que contiene la declaración de la misma.
También puede definirse como un variable local a un método, que es una variable definida dentro del código del método a la que sólo puede accederse desde dentro de dicho código. Otra posibilidad es definirla como parámetro de un método (en este caso llamadas variables de parámetro), que son variables que almacenan los valores de llamada al método y que, al igual que las variables locales, sólo puede ser accedida desde código ubicado dentro del método.

```
//Ejemplo:
namespace T{
   class Test{
   public:
      static void Main(string^[] args){
         string^ e = Console::ReadLine();
         for (int  i=0; i<20; i++){
            Console::WriteLine("la variable local i vale: " + i);
            Console::ReadLine();
         }
         //error porque la variable local i no existe en este contexto;
         int m= i * 2;
         //no se puede asignar valor a la variable
         //local i porque esta declarada dentro del ciclo for.
         i=5;
      }
   }
}
```

En el ejemplo anterior muestra que la variable i es local sólo se la puede utilizar dentro del bloque for en el cual fue declarada.

Variables Estáticas

Un campo miembro es considerado como una variable estática cuando es declarado con el modificador "`static`". Una variable estática pertenece a la clase donde es definida y conserva su valor a lo largo de toda la ejecución del programa.

```
//Ejemplo:
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
```

```
using DotNET::System;

namespace test{
   class A{
      int x;
   public:
      //declaración de la variable estática y;
      static int y;
      A(int a, int b){
         x = a;
         y = b;
      }
   }
   class App{
   public:
      static void Main(string^[] args){
         A^ a = new A(2,3);
         //se accede a la variable estática
         Console::WriteLine("A::y vale : " + A::y);
         Console::ReadLine();
      }
   }
}
```

En el ejemplo anterior se observa que la variable y es estática, es decir pertenece a la clase A y para acceder a la misma es necesario usar el nombre de la clase y el operador "::", en el ejemplo anterior es A.

Variables de Instancia (campos)

Un campo miembro es considerado como una variable de instancia cuando es declarado sin el modificador "`static`". Una variable de instancia comienza su existencia al mismo tiempo en el que una instancia de su tipo contenedor comienza a existir y deja de existir después de que la instancia de su tipo contenedor ejecuta su destructor, si lo hubiere, o es eliminada.
El valor inicial por defecto de toda variable de instancia es el valor por defecto del tipo de la variable de instancia.

```
//Ejemplo:
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   class Alumno{
      //variable de instancia
      string^ nombre;
      //variable de instancia
      string^ apellido;
   public:
       Alumno(string^ nombres, string^ apellidos){
         nombre = nombres;
         apellido = apellidos;
      }
   }
   class Test{
   public:
      static void Main(string^[] args){
         Alumno^ a = new Alumno("Mariano", "Perez");
            Console::WriteLine("se creo el alumno Mariano Perez");
         Console::ReadLine();
      }
   }
```

```
}
```

Variables de Parámetro

Se define variable de parámetro a las variables introducidas por la declaración de parámetros en una función y poseen como alcance todo el cuerpo de la función.
Las variables de parámetro se pueden utilizar en constructores (ejemplo I) y también se las puede utilizar en métodos o indexadores (ejemplo II).
Se puede observar con el siguiente ejemplo que las variables de parámetro es el nombre y apellido pasado al constructor, en este caso es al constructor de Persona.

```
//Ejemplo I:

namespace Test{
   class Persona{
   public:
      string^ Nombre;
      string^ Apellido;
      Persona(string^ nombre, string^ apellido){
         this.Nombre = nombre;
         this.Apellido = apellido;
      }
   }
   class Trabajador inherits Persona{
   public:
      int sueldo;
      Trabajador(string^ nombre,string^ apellido,int sueldo):
         base(Nombre,Apellido)
      {
         //variables de parámetros son los apellido, nombre y sueldo;
         this.sueldo = sueldo;
      }
   }
   public class App{
   public:
      static void Main(string^[] args){
         Trabajador^ p = new Trabajador("belen","perez",2000);
         Console::WriteLine("Se creo el  nuevo trabajador");
         Console::ReadLine();
      }
   }
}

//Ejemplo II:
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
using DotNET::System;

namespace test{
   public class P{
      float promedio(int n1, int n2){
         // Variables de parametro n1 y n2
         float promedios = n1 / n2;
         return promedios;
      }
   }
   class App{
   public:
      static void Main(string^[] args){
         P^ p = new P();
         float b = p.promedio(2, 3);
         Console::WriteLine("el promedio es:" + b);
         Console::ReadLine();
      }
   }
```

```
}
```

Constantes

Una constante es una variable cuyo valor puede determinar el compilador durante la compilación y puede aplicar optimizaciones derivadas de ello. Para que esto sea posible se ha de cumplir que el valor de una constante no pueda cambiar durante la ejecución, por lo que el compilador informará con un error de todo intento de modificar el valor inicial de una constante. Las constantes se definen como variables normales pero precediendo el nombre de su tipo del modificador const y dándoles siempre un valor inicial al declararlas. O sea, con esta sintaxis:

```
   const <tipoConstante> <nombreConstante> = <valor>;
```

Así, ejemplo de definición de constante es el siguiente:

```
   const int a = 123;
   const int b = a + 125;
```

Dadas estas definiciones de constantes, lo que hará el compilador será sustituir en el código generado todas las referencias a las constantes a y b por los valores 123 y 248 respectivamente, por lo que el código generado será más eficiente ya que no incluirá el acceso y cálculo de los valores de a y b. Nótese que puede hacer esto porque en el código se indica explícitamente cual es el valor que siempre tendrá a y, al ser este un valor fijo, puede deducir cuál será el valor que siempre tendrá b. Debido a la necesidad de que el valor dado a una constante sea precisamente constante, no tiene mucho sentido crear constantes de tipos de datos no básicos, pues a no ser que valgan null sus valores no se pueden determinar durante la compilación sino únicamente tras la ejecución de su constructor.

```
//Ejemplo de Utilización de constante:

import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   public class Test{
   public:
      static void Main(string^[] args){
         const int a=3;
         Console::WriteLine("el valor de la constante es:" +  a );
         Console::ReadLine();
      }
   }
}
```

2.2 Tipos de Datos

Los Tipos Básicos o Nativos son la base del sistema de tipos y son proporcionados por el lenguaje. Los Tipos Básico o Nativos son:

- ✓ Enteros con signo: sbyte, short, int, long
- ✓ Enteros sin signo: byte, ushort, uint, ulong
- ✓ Punto Flotante: float, doble, decimal
- ✓ Carácter: char

Los Tipos Definidos por el Usuario son los tipos definibles por el usuario en combinación de los tipos básicos proporcionados intrínsecamente por el lenguaje. Dentro de esta clase de tipos de datos se encuentran los siguientes:

- ✓ Clases
- ✓ Interfaces
- ✓ Estructuras

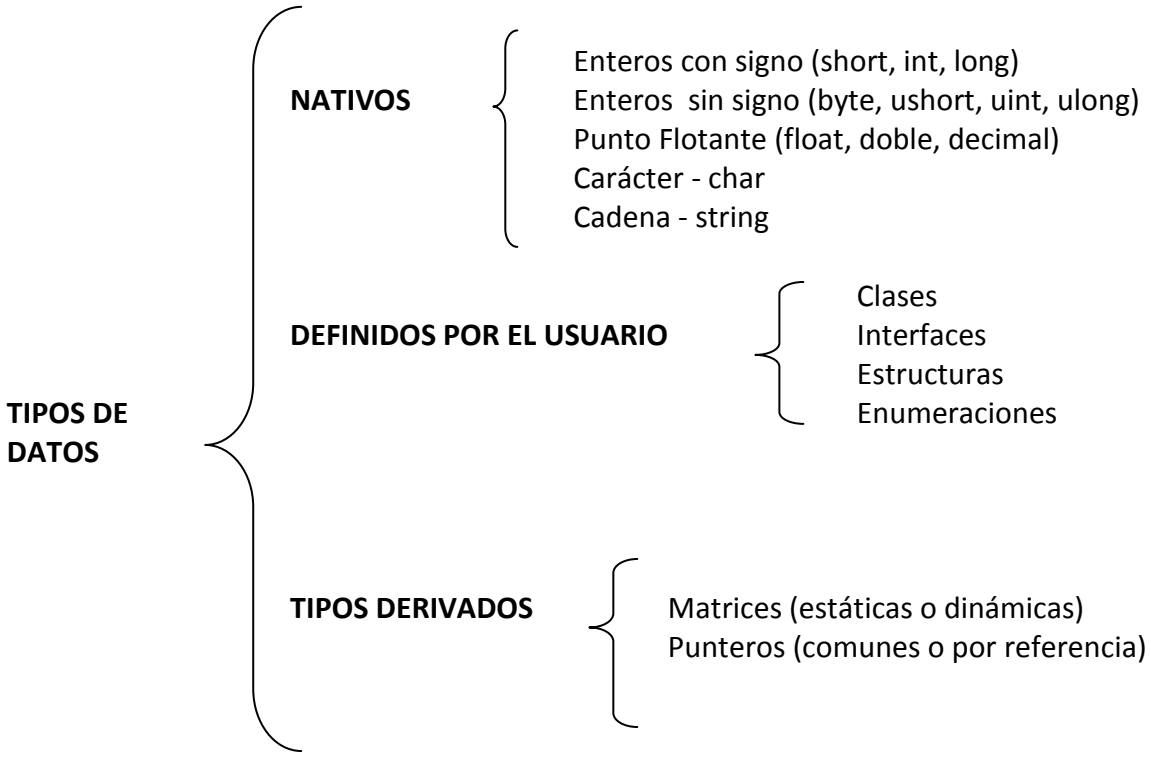
Los Tipos Derivados son modificaciones de tipos de datos existentes, como punteros y matrices. Zoe proporciona un sistema de tipos unificado y orientado a objetos, cada variable de cualquier tipo de datos puede ser tratado como un objeto sin importar si es un Tipo Básico o un tipo definido por el Usuario.

Los Tipos Derivados son los siguientes:

- ✓ Matrices
- ✓ Punteros

Por defecto todo tipo de datos deriva explícita o implícitamente del tipo “object”, el tipo “object” es la clase base final de todos los tipos en la configuración por defecto del código Zoe.

A continuación se definen los tipos básicos, luego las clases de tipos definidos por el usuario, los tipos derivados y finalmente se definen las reglas del sistema unificado de tipos de Zoe.



2.3 Tipos de Almacenamiento

Meta D++ duplica los tipos de almacenamiento de Zoe, el cual define tres clases de almacenamiento bien diferenciados para las variables, estos son:

- ✓ Almacenamiento de Pila
- ✓ Almacenamiento de Montón Recolectable
- ✓ Almacenamiento de Montón No Recolectable

En el Almacenamiento de Pila las variables son almacenadas en la propia pila de llamadas del programa, por lo que toda variable de pila posee su existencia limitada como máximo por la función en la que es declarada. El Almacenamiento de Pila se exige que se encuentre disponible de forma limitada en todas las plataformas de implementación.
En el Almacenamiento de Montón Recolectable las variables son almacenadas en un espacio de memoria asignado por el entorno de tiempo de ejecución (runtime) de la plataforma de implementación en uno o más bloques de memoria, todo espacio de almacenamiento asignado dinámicamente en estos bloques de memoria es liberado automáticamente por el recolector de basura o "garbage collector". El Almacenamiento de Montón Recolectable se exige sea implementado por todas las plataformas de implementación.
El Almacenamiento de Montón No Recolectable es al igual que el Almacenamiento de Montón Recolectable uno o más bloques de memoria asignados por el entorno de tiempo de ejecución de la plataforma de implementación, con la diferencia de que la memoria asignada en estos bloques de memoria debe ser liberada explícitamente por el programa Zoe (y por tanto por el programa Meta D++). El Almacenamiento de Montón No Recolectable no se requiere que sea implementado por todas las plataformas de implementación, sólo los Módulos de Salida Zoe Clases A y R requieren implementar almacenamiento de montón no recolectable, por lo que se recomienda que su utilización sea limitada sólo a casos especiales donde es realmente necesario y beneficioso.

2.2.1 Nativos

Los tipos básicos disponibles en Meta D++ (equivalentes a los tipos básicos Zoe) son los siguientes:
El Lenguaje LayerD D++ Tipos

- ✓ Enteros con signo: "sbyte", "int", "short", "long"
- ✓ Enteros sin signo: "byte", "unsigned", "usigned short" ó "ushort", "unsigned
- ✓ long" ó "ulong"
- ✓ Reales de coma flotante: "float", "double", "long double"
- ✓ Otros reales: "decimal"
- ✓ De carácter: "char"
- ✓ De cadena: "string"
- ✓ Otros: "void", "object", "bool"
- ✓ De Classfactorys: "iname", "expression", "block", "`type`"

Los tipos de entero con signo ordenados por rango de valores soportados son: "`sbyte`", "`int`", "`short`", "`long`".

El tipo "`sbyte`" representa un entero con signo de 8bits y el tipo "`sbyte`" es el único tipo entero con signo de igual longitud sin importar la plataforma de implementación.
Los tipos enteros "short", "integer" y "long" tienen una longitud máxima que depende de la implementación del modulo de salida Zoe utilizado. El tipo "integer" representa un entero típico en la plataforma de destion, por ejemplo de 32bits en una plataforma de 32bits o de 64bits en una plataforma de 64bits. El tipo "short" representa un entero más pequeño que "integer" y "long" representa un entero de mayor amplitud que un "integer". Para más información vea la Especificación de Meta D++.

Enteros sin Signo

Los tipos de entero sin signo ordenados por rango de valores soportados son: "`byte`", "`unsigned`", "`unsigned short`" ó "`ushort`", "`unsigned long`" ó "`ulong`".
Los Enteros Sin Signo no se requiere que sean soportados por todas las plataformas de destino, sin embargo es recomendable.
El tipo "`byte`" representa un entero sin signo de 8bits y soporta el rango de decimales del 0 al 255.

Reales de Coma Flotante

Los tipos de números reales con coma flotante disponibles en Meta D++ son "`float`", "`double`" y "`long double`".
El tipo "`float`" debe corresponderse en una determinada plataforma de implementación con el formato de precisión simple (32bits)
El tipo "`double`" debe corresponderse en una determinada plataforma de implementación con el formato de doble precisión (64bits). En caso de que la plataforma proporcione una precisión mayor para el tipo "`float`", la precisión del tipo "`double`" siempre debe ser mayor que la precisión del tipo "`float`" implementado.
El tipo "`long double`" es opcional a ser implementado en una determinada plataforma de implementación por un módulo de salida. En caso de ser implementado se requiere tenga una precisión mayor al tipo "`double`", si no es implementado se debe utilizar la misma precisión que la utilizada para el tipo "`double`" y emitir "avisos" (warnings) indicando la disminución de precisión.
El tipo "`decimal`" representa un número real que se recomienda duplique la precisión del tipo de coma flotante "`double`", sin embargo se deja a cada módulo de salida la decisión sobre implementarlo usando coma flotante o punto fijo.
En caso de no implementar un tipo especial, el módulo de salida debe proporcionar para "`decimal`" la misma implementación que para el tipo "`double`" y emitir un "aviso" (warning) para el usuario.

De Carácter

Los tipos de datos de carácter disponibles en Zoe son "`char`".
El tipo "`char`" representa un único carácter y por defecto posee una implementación de 16bits. El operador binario "`+`" entre dos operandos de tipo "`char`" devuelve un tipo "`string`" con la concatenación de ambos operandos.
Se representa con la clase zoe::lang::Char.

De Cadena

Zoe define el tipos de datos de cadena estándar "`string`", el cual se corresponde con las clase "`zoe::lang::String`".
El tipo "`string`" representa una secuencia de caracteres inmutable, por tanto toda operación en un string devuelve un nuevo valor.
Entre otras funciones las clases de expresión deben proporcionar una implementación para el operador binario "`+`" que funcione de la siguiente manera:
• Si los dos operadores son de tipo "`string`" el resultado es la concatenación de ambas cadenas también del tipo "`string`".
• Los operadores relacionales devuelven el tipo "`bool`".

Otros

Los tipos "`void`", "`object`", "`bool`" de Zoe se definen a continuación.
El tipo "`void`" representa el tipo "no especificado" ó "cualquier tipo", puede utilizarse como tipo de retorno en funciones indicando que la función no devuelve ningún valor ó se puede utilizar como puntero a una variable de cualquier tipo, vea punteros.
El tipo "`object`" es una abreviación para la clase "`zoe::lang::Object`". Por defecto todo tipo de datos deriva explícita o implícitamente del tipo "`object`", el tipo "`object`" es la clase base final de todos los tipos en la configuración por defecto del código Zoe. Opcionalmente se puede especificar que esto no se cumpla en la sección de configuración de un Documento Zoe, para más datos vea la especificación de Zoe. En Meta D++ no existe una sintaxis para desactivar el sistema de tipado unificado, sin embargo es posible hacerlo escribiendo un archivo de configuración Zoe y pasándolo como parámetro a una implementación del compilador Zoe al compilar el programa.
Por defecto el tipo "`object`" posee una interfaz definida por el lenguaje Zoe. En Meta D++ no se define una sintaxis para cambiar la implementación de la clase `zoe::lang::Object`, pero es posible hacerlo escribiendo a mano un archivo de configuración Zoe.
El tipo "`bool`" representa un valor lógico "verdadero" (`true`) ó "falso" (`false`).
Los valores permitidos para "`bool`" son "`true`" y "`false`".
No se definen conversiones estándar entre el tipo "`bool`" y otros tipos como los tipos enteros.
El tamaño del tipo "`bool`" es dependiente de la plataforma de implementación, pero se recomienda utilizar un byte para su representación.
El tipo "`bool`" no puede ser usado en lugar de un tipo entero o viceversa.

2.2.2 Tipos Especiales

Zoe define tres tipos especiales que son accesibles sólo por classfactorys, estos tipos son el tipo "`iname`", el tipo "`expression`", el tipo "`block`" y el tipo "`type`".

2.3.1 Tipos Definidos por el Usuario

Clases, Interfaces, Estructuras y Enumeraciones

Zoe y por tanto Meta D++ proporciona cuatro clases de tipos definibles por el usuario a partir de la combinación de Tipos Básicos, Tipos Derivados y Otros Tipos Definidos por el Usuario, estas cuatro clases de tipos son: las clases, las interfaces, las estructuras y las enumeraciones. Para más información de cualquiera de estos cuatro tipos de clases vea los capítulos correspondientes.

2.3.2 Tipos Derivados

2.3.2.1 Punteros

Los punteros son un tipo derivado a partir de otros tipos, se pueden declarar punteros a cualquier tipo en Meta D++, como punteros a matrices, punteros a tipos básicos, punteros a punteros, punteros a función miembro, etc.

```
//puntero a entero
int *var;
//un puntero a un puntero (dos niveles de indirección)
int **var;
```

2.3.2.2 Punteros Constantes y Volátiles

Se puede modificar la semántica de los punteros estableciendo los atributos "`const`" y "`volatile`" en su declaración.
Un puntero "`const`" ó constante debe ser inicializado en su declaración y no permite luego que su valor sea sobrescrito.
Un puntero "`volatile`" ó volátil indica al compilador, y más precisamente a los Módulo de Salida Zoe, que no se optimice el acceso al puntero y que su valor sea recuperado de la memoria cada vez que es requerido. Esto disminuye el rendimiento del puntero, pero a veces es la única opción cuando se trabaja con datos que pueden ser accedidos por otros procesos o interrupciones de hardware.
Para declarar un puntero, es decir el valor del puntero, como "`const`" o "`volatile`" utilice la sintaxis siguiente:

```
{
  int *const punteroConstante;
  int *volatile punteroVolatil;
}
```

Un puntero declarado como "`const`" se dice que es un *puntero constante*.
Un puntero declarado como "`volatile`" se dice que es un *puntero volátil*.
Un puntero que no es un *puntero constante* (no se utilizo "`const`" en su declaración) se dice que es un *puntero no constante*.
Para indicar las características de "`const`" y "`volatile`" del dato al cual apunta el puntero (no del puntero mismo) la sintaxis es la siguiente:

```
{
  const int *punteroADatoConstante;
  volatile int *punteroADatoVolatil;
```

```
}
```

Un puntero que apunta a un dato constante se dice que es un *puntero a dato constante*.
Un puntero que apunta a un dato volátil se dice que es un *puntero a dato volátil*.
Un puntero que no es un *puntero a dato constante* es un *puntero a dato no constante*.
Un *puntero a dato constante* o un *puntero a dato volátil* pueden a su vez ser *puntero constante* y/o *puntero volátil*.
Las siguientes restricciones se aplican a *punteros a dato constante*:
Un *puntero a dato no constante* no puede asignarse con la dirección de una variable constante.
Un *puntero a dato constante* puede asignarse con la dirección de una variable no constante o una variable constante.
Se proporciona una conversión implícita para todo tipo "`T*`" al tipo "`const T*`". La conversión inversa no existe implícitamente.
Las restricciones de los *punteros a dato constante* son validas para *punteros a dato volátil* o una combinación de ellos; entendiéndose que no es posible asignar un puntero a dato no volátil con la dirección de una variable volátil.
Además a los *punteros constantes* se aplican las siguientes restricciones:
Un *puntero constante* puede asignarse a un *puntero no constante*.
Un *puntero no constante* no puede asignarse a un *puntero constante*.
A continuación se muestran ejemplos de declaraciones y asignaciones válidas e invalidas de punteros constantes y punteros a constantes:

```
{
  char DatoNoVolatilNoConstante='A';
  const char DatoNoVolatilConstante='B';
  volatile char DatoVolatilNoConstante='C';
  volatile const char DatoVolatilConstante='D';

  char *PunteroADatoNoVolatilNoConstante;
  const char *PunteroADatoNoVolatilConstante;
  volatile char *PunteroADatoVolatilNoConstante;
  volatile const char *PunteroADatoVolatlConstante;

  PunteroADatoNoVolatilNoConstante=&DatoNoVolatilNoConstante; //OK
  PunteroADatoNoVolatilConstante=&DatoNoVolatilConstante; //OK
  PunteroADatoVolatilNoConstante=&DatoVolatilNoConstante; //OK
  PunteroADatoVolatlConstante=&DatoVolatilConstante; //OK

  PunteroADatoNoVolatilNoConstante=&DatoVolatilConstante; //Error
  PunteroADatoNoVolatilConstante=&DatoVolatilNoConstante; //Error
  PunteroADatoVolatilNoConstante=&DatoNoVolatilNoConstante; //OK
  PunteroADatoVolatilConstante=&DatoVolatilNoConstante; //Error

  char *const PunteroConstante=&DatoNoVolatilNoConstante; //OK
  char *const PunteroConstante=&DatoNoVolatilConstante; //Error
  char *volatile PunteroVolatil=&DatoNoVolatilNoConstante; //OK
  char *volatile PunteroVolatil=&DatoVolatilNoConstante; //OK
  const char *const PunteroConstante=&DatoNoVolatilConstante; //Error

  PunteroConstante=PunteroADatoNoVolatilNoConstante; //Error
  PunteroADatoNoVolatilNoConstante=PunteroConstante; //OK
}
```

2.3.2.3 Matrices

Una matriz es una estructura de datos que permite acceder a múltiples variables utilizando un solo identificador y un índice.

Una matriz es estática cuando se define la cantidad de elementos antes de la ejecución del programa.
La sintaxis de definición de matrices es la siguiente:

```
<tipodedatosdelelemento>[] <nombrematriz> = new <tipodato> [<cantidad de elementos>];
Ejemplo:
   int [] var = new  int[10];
```

El tipo de datos puede ser entero (int), string, double, o también puede ser una referencia a una clase.

```
Ejemplo:
   MiClase^[] var = new  MiClase^[10];
   string^[] v = new string^[10];
```

En los ejemplos definidos anteriormente se observa que la dimensión de esas matrices es sólo una, esto se refleja en la cantidad de pares de corchetes que tiene en la sintaxis de la misma
Para matrices de más de una dimensión la sintaxis es la siguiente:

```
 <tipo de dato> [][] <variable> = new <tipodato>[<cantidad de elementos>];
```

Para crear una matriz unidimensional se pueden seguir los siguientes pasos:
1. Crear la ***referencia*** indicando con un doble corchete que es una *referencia a matriz*,
`int[] mat;`
2. Crear el vector de referencias a las filas, lo cual puede hacerse antes de la ejecución o bien puede ser en el momento de la ejecución de una aplicación o método
`mat = new int[nfilas];`

A continuación se presenta un ejemplo de creación de matriz unidimensional, cuya cantidad de elementos se asigna antes de realizar la ejecución de la aplicación

```
// crear una matriz
// se inicializan a cero
double mat[] = new double[2];
int [] b = {{1, 2}};
```

```
//Ejemplos:

import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   class Test{
   public:
      static void Main(string^[] args){
         int[] matriz = new int[] = {1,2,3,4};
         for (int  i=0;i < matriz.Length;i++){
            Console::WriteLine("el valor de elemento es:" + i);
            Console::ReadLine();
         }
      }
   }
}
```

En el ejemplo siguiente se mostrará como se realiza una matriz dinámica, es decir aquella que se le puede indicar la cantidad de elementos que debe tener en el momento de ejecutar la

aplicación y de esta forma asegurar de utilizar la cantidad de memoria adecuada y no desperdiciar el espacio de la misma.

```
//Ejemplo:
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;
using DotNET::System;

namespace T{
   public class Test{
   public:
      static void Main(string^[] args){
         Console::WriteLine("Ingrese la cantidad de elementos  de la matriz:");
         string^ e = Console::ReadLine();
         int n = Convert::ToInt32(e);
                  //asignación del tamaño de la matriz en el momento de la ejecución
         int[] matriz = new int[n];
         for (int  i=0;i < matriz.Length;i++){
            matriz[i] = i;
            Console::WriteLine("el valor de elemento " + i + "es:" + i);
            Console::ReadLine();
         }
      }
   }
}
```

También es posible definir matrices multidimensionales. En el caso de este tipo de matrices, también es posible indicar la cantidad de elementos en el momento de la ejecución.
A continuación vemos un ejemplo de matrices multidimensionales:

```
//Ejemplo:
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   class Test{
   public:
      static void Main(string^[] args){
         int[][] matriz = new int[5][];
         for (int  i=0;i < 5;i++){
            //Asigno espacio de memoria para  matriz
            matriz[i] = new int[5];
         }
         for (int  n=0;n < 5;n++){
            for (int  j=0;j < 5;j++){
               //Cargar matriz;
               matriz[n][j] = j;
               Console::WriteLine("n,j vale:" + matriz[n][j]);
               Console::ReadLine();
            }
         }
      }
   }
}
```

3 OPERADORES

Un operador es un símbolo formado por uno o más caracteres que permite realizar una operación en una, dos o tres expresiones y producen un resultado. Meta D++ permite sobrecargar algunos de los operadores en las clases definidas por el usuario.

3.1 Operadores Aritméticos

Los Operadores Aritméticos son:

- ✓ Suma (+)
- ✓ Resta (-)
- ✓ Multiplicación (*)
- ✓ División (/)
- ✓ Módulo (%)

```
// Ejemplos de utilización del operaciones aritméticas:

import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   class Test{
   public:
      static void Main(string^[] args){
         int suma = 1;
         float division = 1000;
         for (int i=1;i < 20;i++){
            suma = suma + i;
            division = division / i;
            Console::WriteLine("el resultado  es :" + division);
         }
         Console::WriteLine("el valor total de la suma es:" + suma);
         Console::ReadLine();
      }
   }
}
```

3.2 Operadores Lógicos:

Los operadores lógicos son los siguientes:

- ✓ and (&& y &)
- ✓ or (|| ó |),
- ✓ not(!)
- ✓ xor (or exclusive)

Los operadores && y || se diferencian de & y | en que los primeros solo evalúan el segundo operando si el primero es verdadero, en cambio los operadores & y | siempre evalúan los dos operandos.

A continuación se mostrará la tabla de verdad de los operadores lógicos mencionados anteriormente.

Tabla de Verdad para el Operador Lógico AND (&&)

And	True	False
True	True	False
False	False	False

Como puede apreciarse en la tabla anterior, para que el resultado sea verdadero (true) los dos operando deben ser verdaderos.

Tabla de Verdad para el Operador OR

OR	True	False
True	True	True
False	True	False

En el caso del operador OR el resultado será verdadero (true) cuando alguno de los operandos sea verdadero.

Tabla de Verdad para el Operador NOT

NOT	
True	False
False	True

Como puede visualizarse en la tabla de verdad del operador lógico NOT el resultado, después de aplicar dicho operador será invertido, es decir si el operando es true el resultado después de aplicado dicho operador será false.

Tabla de Verdad para el Operado XOR

XOR	True	False
True	True	False
False	False	True

En el caso de este tipo de Operador, el resultado será verdadero cuando ambos operando sean iguales, es decir el resultado será true cuando ambos miembros sean verdaderos o falsos, pero ambos.

```
//Ejemplo de utilización del operador and (&&)
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   class Test{
   public:
      static void Main(string^[] args){
         int suma = 0;
         Console::WriteLine("Ingrese un número:");
         string^ e = Console::ReadLine();
         int n = Convert::ToInt32(e);
```

```
        if (n > 2 && n < 20) //utilización de &&
                    {
           Console::WriteLine("el numero es ingresado correctamente");
        }
        else {
           Console::WriteLine("el numero  ingresado  no es correcto, por favor ingrese
el numero adecuado");
        }
     }
   }
}
```

```
//Ejemplo de utilización del operador or (||)

import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   class Test{
   public:
      static void Main(string^[] args){
         string^ e = Console::ReadLine();
         int n = Convert::ToInt32(e);
         if (n == 2 || n == 5){
            Console::WriteLine("el numero ingresado no es correcto");
            return ;
         }
      }
   }
}
```

```
//Ejemplo de Utilización del operador  not(!)

import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   class Test{
   public:
      static void Main(string^[] args){
         string^ e = Console::ReadLine();
         int n = Convert::ToInt32(e);
         if (n != 5) //utilizaciòn del operador distinto
                     {
            Console::WriteLine("el numero ingresado no es correcto");
            return ;
         }
      }
   }
}
```

3.3 Operadores de Comparación u Operadores Relaciónales

Operación	Resultado
==	Compara la igualdad del valor de dos expresiones
!=	Se utiliza para consultar si el valor de una expresión es distinto del valor de otra expresión

<=	Compara si el valor de una expresión es menor o igual que el valor de otra expresión
>=	Compara si el valor de una expresión es mayor o igual que el valor de otra expresión
>	Compara si el valor de una expresión es mayor que el valor de otra expresión
<	Compara si el valor de una expresión es menor que el valor de otra expresión

Si uno de los operandos en un operador binario es de tipo de coma flotante el otro tipo debe ser un entero u otro tipo de coma flotante. En cuyos casos se debe proceder como sigue:

• Si el operando es de tipo entero se convertirá al mismo tipo de coma flotante del otro operando.

• Si los dos operandos son del mismo tipo de coma flotante la operación se realizará utilizando dicho tipo y el resultado será en el mismo tipo que el de los operandos.

• Si los dos operandos son de tipo de coma flotante pero de diferente precisión, se convertirán los operandos de menor precisión al tipo del operando de mayor precisión, la operación se realizará con el tipo de mayor precisión (o una precisión mayor) y el resultado será del tipo de mayor precisión.

• Para los operadores relacionales (“==”, “!=”, “<=”, “>=”, “<”, “>”) el tipo del resultado será “bool”.

```
//Ejemplo de Utilización de operador < y operador >
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
using DotNET::System;

namespace T{
   class Test{
   public:
      static void Main(string^[] args){
         int suma = 0;
         string^ e = Console::ReadLine();
         int n = Convert::ToInt32(e);
         if (n > 2 && n < 20){
            //utilización del operador < y >
            Console::WriteLine("el numero es ingresado correctamente");
            Console::ReadLine();
         }
         else {
            Console::WriteLine("el  nro ingresado  no escorrecto,   favor ingrese el
numero adecuado");
            Console::ReadLine();
         }
      }
   }
}
```

```
//Ejemplo de utilización del Operador (!=)
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
using DotNET::System;

namespace T{
   public class Test{
   public:
      static void Main(string^[] args){
         int suma = 0;
         string^ e = Console::ReadLine();
         int n = Convert::ToInt32(e);
```

```
        //el numero tiene que ser 2 para que sea correcto
        if (n != 2)
        {
           //utilización del operador
           Console::WriteLine("el numero es ingresado correctamente");
           Console::ReadLine();
        }
        else {
           Console::WriteLine("el  nro ingresado  no es correcto,   favor ingrese el
numero adecuado");
           Console::ReadLine();
        }
     }
   }
}
```

3.4 Operaciones con Cadenas:

Cada operación que se realice con cadenas genera una copia de la cadena original, pero nunca modifica a la variable original. Así por ejemplo, si tenemos una cadena, cuyo contenido es "Mariano" y le aplicamos el método ToUpper, se generará una nueva cadena con el resultado del método ToUpper, es decir el nuevo valor será "MARIANO".

Para el manejo de cadenas se puede usar los siguientes operadores y métodos:

- Operador de concatenación (+)
- Operador de indización ([])
- ToString
- IndexOf
- Substring
- ToUpper
- ToLower
- Trim
- LastIndexOf

```
//Ejemplo de Concatenación de cadenas:
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
using DotNET::System;

namespace T{
   public class Test{
   public:
      static void Main(string^[] args){
         string^ expresionentrada = "Bienvenidos";
         string^ expresionsegundaparte = " a Meta D++";
         //utilización del operador de concatenación
         string^ expresionresultado = expresionentrada +
                           expresionsegundaparte;
         Console::WriteLine("el resultado es:" + expresionresultado);
         Console::ReadLine();
      }
   }
}
```

En este ejemplo puede visualizarse que el operador concatenación (+) realiza la unión entre el contenido de dos cadena. La cadenaentrada almacena la expresión "Bienvenidos", la cadena segundaparte almacena " a Meta D++" y la cadena resultado es la unión entre las dos cadenas mencionadas anteriormente, luego de haber aplicado el operador de concatenación. El resultado de la última línea del código anterior es: "Bienvenidos a Meta D++".

```
//Ejemplo de Utilización del Operador de Indización ([])
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
using DotNET::System;

namespace T{
   public class Test{
   public:
      static void Main(string^[] args){
         string^ cadenaminuscula = "belen";
         char caracter = cadenaminuscula[2];
         Console::WriteLine("El carácter es :" + caracter);
         Console::ReadLine();
      }
   }
}
```

En el ejemplo anterior se puede visualizar la funcionalidad del operador de indización, el cual guarda en "caracter" la letra l, es decir la posición que se indica después de cadenaminuscula contando desde la posición cero.

```
//Ejemplo de Utilización de ToUpper
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
using DotNET::System;

namespace T{
   class Test{
   public:
      static void Main(string^[] args){
         string cadenaminuscula = "belen";
         string cadenamayuscula = cadenaminuscula.ToUpper();
         Console::WriteLine("el resultado es:" + cadenamayuscula);
         Console::ReadLine();
      }
   }
}
```

Con el ejemplo de arriba muestra la funcionalidad de ToUpper, el cual convierte una cadena que está en minúscula a mayúscula.

```
//Ejemplo de utilización de ToLower
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
using DotNET::System;

namespace T{
   class Test{
   public:
      static void Main(string^[] args){
         string cadenaminuscula = "BELEN";
         //utilización de ToLower
         string cadenamayuscula = cadenaminuscula.ToLower();
         Console::WriteLine("el resultado es:" + cadanamayuscula);
         Console::ReadLine();
      }
   }
}
```

ToLower es la función inversa a ToUpper, convierte de mayúsculas a minúsculas.

```
//Ejemplo de utilización de IndexOf
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
using DotNET::System;
```

```
namespace T{
   class Test{
   public:
      static void Main(string^[] args){
         string^ expresion2 = "e";
         string^ expresion1 = "belen";
         //utilización de IndexOf
         int indice = expresion1.IndexOf(expresion2);
         Console::WriteLine(indice);
                     Console::ReadLine();
      }
   }
}
```

`IndexOf` indica cuál es el índice de la primera aparición de la expresión2 (que en el ejemplo anterior es e) en la expresión1 (belen). La búsqueda de dicha subexpresión expresión2 se realiza desde el principio de la expresión1, pero es posible indicar en un segundo parámetro opcional de tipo `int` cuál es el índice a partir del que se desea empezar a buscar. Del mismo modo, la búsqueda acaba al llegar al final de la expresión sobre la que se busca. Si no se encuentra la subcadena o carácter la función retorna -1.

```
//Ejemplo de utilización de Substring
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
using DotNET::System;

namespace T{
   class Test{
   public:
      static void Main(string^[] args){
         string^ expresion1 = "belen";
         string^ expresionresultado = expresion1.Substring(0, 3);
         Console::WriteLine("El resultado es:" + expresionresultado);
      }
   }
}
```

`string Substring(int posicion, int numeroCaracteres)`: Devuelve la porción de la cadena sobre la que se aplica que comienza en la posición indicada y tiene el número de caracteres especificados. Si no se indica dicho número se devuelve la subcadena que va desde la posición indicada hasta el final de la cadena. En el ejemplo anterior se aplico el Substring a expresion1 (belen) desde la posición cero y a partir de allí copiar 3 (tres) caracteres. Luego el resultado que muestra es “bel”.

```
//Ejemplo de utilización de Trim
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
using DotNET::System;

namespace T{
   class Test{
   public:
      static void Main(string^[] args){
         string^ nombre = " belen";
         Console::WriteLine("el nombre antes de aplicar trim :" + nombre);
         Console::ReadLine();
         string^ resultado = nombre;
         resultado.Trim();
         Console::WriteLine("el resultado despues de aplicar Trim :" + resultado);
      }
   }
}
```

```
//Ejemplo de Utilización de LastIndexOf
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
using DotNET::System;

namespace T{
   class Test{
   public:
      static void Main(string^[] args){
         string^ nombre= "belen";
         int resultado = nombre.LastIndexOf('e');
         Console::WriteLine("La última e se encuentra en: "  + resultado);
         Console::ReadLine();
      }
   }
}
```

Esta función devuelve la última posición de la letra o cadena que se indica después como argumento.

3.5 Operaciones con Fechas

Para manipular fechas y horas puede utilizar el tipo de datos nativo `zoe::lang::DateTime`. Meta D++ proporciona ciertas funciones para el manejo de las fechas. Las cuales se mencionan a continuación:

```
DateTime(int month, int day, int year);
override string^ ToString();
static const DateTime MinValue = new DateTime(1,1,1000);
static const DateTime MaxValue = new DateTime(1,1,9999);
static const int Size;

DateTime property Now{
   get;
}
int property Day{
   get;
   set;
}
int property Month{
   get;
   set;
}
int property Year{
   get;
   set;
}
int property Hour{
   get;
   set;
}
int property Minutes{
   get;
   set;
}
int property Seconds{
   get;
   set;
}
void AddDays(int days);
void AddMonths(int months);
void AddYears(int years);
void AddHours(int hours);
void AddMinutes(int minutes);
void AddSeconds(int seconds);
int CompareTo(DateTime opA);
```

```
// Operadores para Fechas
static TimeSpan operator+(DateTime opA, DateTime OpB);
static TimeSpan operator-(DateTime opA, DateTime OpB);
static bool operator==(DateTime opA, DateTime OpB);
static bool operator!=(DateTime opA, DateTime OpB);
static bool operator>(DateTime opA, DateTime OpB);
static bool operator<(DateTime opA, DateTime OpB);
static bool operator>=(DateTime opA, DateTime OpB);
static bool operator<=(DateTime opA, DateTime OpB);
```

Algunos ejemplos:

```
//Ejemplo de utilización de AddDays
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
using DotNET::System;
using zoe::lang;

namespace test{
   public class App{
   public:
      static void Main(string^[] args){
         DateTime Fecha = new DateTime(4,30,2008);
         Console::WriteLine("fecha origen:" + Fecha.ToString());
         Fecha.AddDays(2);
         Console::WriteLine("la fecha resultante, despues de sumar 2 dias:" +
Fecha.ToString());
      }
   }
}
```

```
//Ejemplo de utilización de AddHours
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
using DotNET::System;
using zoe::lang;

namespace test{
   public class App{
   public:
      static void Main(string^[] args){
         DateTime horaingreso = new DateTime(4,0,0);
         Console::WriteLine("hora de ingreso:" + horaingreso.ToString());
         horaingreso.AddHours(6);
         Console::WriteLine("la  hora de salida es:" + horaingreso.ToString());
      }
   }
}
```

```
//Ejemplo de utilización de AddMonths
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
using DotNET::System;
using zoe::lang;

namespace test{
   public class App{
   public:
      static void Main(string^[] args){
         DateTime fechaingreso = new DateTime(4,0,0);
         Console::WriteLine("la fecha de ingreso de un nuevo curso:" +
fechaingreso.ToString());
         fechaingreso.AddMonths(3);
         Console::WriteLine("la  fecha de salida del curso es:" +
fechaingreso.ToString());
      }
   }
}
```

```
//Ejemplo de Utilización de AddYears
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
using DotNET::System;
using zoe::lang;

namespace test{
   public class App{
   public:
      static void Main(string^[] args){
         DateTime fechaingreso = new DateTime(5,5,2008);
         Console::WriteLine("la fecha de ingreso es:" + fechaingreso.ToString());
         fechaingreso.AddYears(3);
         Console::WriteLine("la  fecha de salida es de :" + fechaingreso.ToString());
      }
   }
}
```

```
//Ejemplo de Utilización de CompareTo
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
using DotNET::System;
using zoe::lang;

namespace test{
   public class App{
   public:
      static void Main(string^[] args){
         DateTime fechaingreso = new DateTime(5,5,2008);
         DateTime fechaegreso = new DateTime(5,5,2008);
         int resultado = fechaingreso.CompareTo(fechaegreso);
         if (resultado == 0){
            Console::WriteLine("las dos fechas son iguales");
         }
      }
   }
}
```

3.6 Operador Condicional:

```
<Condición> ? < expresión1>: <expresión2>
```

El significado del operador es el siguiente: se evalúa <condición>. Si es cierta se devuelve el resultado de evaluar <expresión1> y si es falsa se devuelve el resultado de evaluar la condición2.

Ejemplo:

```
B = (a >0) ? a:0;
```

En este ejemplo, si el valor de la variable a es superior a cero se asignará a b el valor de a, mientras que en caso contrario el valor que se asignará será cero.

```
//Ejemplo de utilización del Operador condicional:
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
using DotNET::System;

namespace T{
   public class Test{
   public:
      static void Main(string^[] args){
         int suma = 0;
         string^ e = Console::ReadLine();
```

```
        int n = Convert::ToInt32(e);
                    //utilización del  operador condicional
        int c= (n > 2)? 2:0;
        Console::WriteLine("el nro resultante es" + c);
        Console::ReadLine();
      }
   }
}
```

3.7 Operaciones con Punteros

Un puntero es una variable que almacena una referencia a una dirección de memoria. Para obtener la dirección de memoria de un objeto se usa el operador &, para acceder al contenido de la dirección de la memoria almacenada en un puntero se usa el operador *, para acceder a un miembro de un objeto cuya dirección se almacena en un puntero se usa -> y para referenciar una dirección de memoria de forma relativa a un puntero se le aplica el operador [] de la forma puntero[desplazamiento].
Para utilizar punteros de sin tener que usar el operador "->" para acceder a los miembros de una clase puede utilizar punteros con semántica de referencia usando el modificador de declaración "^" o "*ref". Por ejemplo:

```
// pointer with reference semantic
Form^ form1 = new MyWindow();
form1.Text = "Hello";

// equivalent declaration
Form *ref form1 = new MyWindow();
```

3.6.1 Expresiones de Conversión de Punteros

Las conversiones implícitas permitidas para los punteros son las siguientes:

Todo puntero puede convertirse implícitamente a un puntero de tipo "`void*`".
El valor "`null`" puede convertirse a cualquier tipo de puntero implícitamente.

Las conversiones explicitas permitidas para los punteros son las mostradas a continuación:

Todo puntero de tipo "`void*`" puede convertirse a cualquier tipo de puntero si se proporciona una conversión explícita, no se permite la conversión implícita del tipo "`void*`" a otro tipo.
La conversión de un puntero a "`void*`" y luego al tipo original del puntero debe conservar sin cambios el puntero obtenido respetando la identidad "`P=(T*)(void*)P`" siendo "`P`" un puntero de tipo "`T*`".
Toda expresión integral puede convertirse a un tipo puntero si se proporciona una conversión explicita. En caso de que el integral no entre en el tipo del puntero se producirá un error en tiempo de compilación o una excepción en tiempo de ejecución del tipo "`zoe::lang::OverflowException`".
Todo puntero puede convertirse a un tipo integral si se proporciona una conversión explícita. En caso de que el puntero no entre en el tipo del integral se producirá un error en tiempo de compilación o una excepción en tiempo de ejecución del tipo "`zoe::lang::OverflowException`".
La conversión de un puntero a un tipo entero y luego al tipo original del puntero debe conservar sin cambios el puntero obtenido respetando la identidad "`P=(T*)(Tipo-Integral*)P`" siendo "`P`" un puntero de tipo "`T*`".

Al convertir un puntero de tipo "`T*`" a un puntero "`void*`" y luego al tipo "`T'*`" el puntero resultante puede referenciar una dirección de memoria no alineada con el tipo de datos "`T'`", en dicho caso las operaciones de indirección y adición sobre el puntero obtenido presentaran resultados indeterminados a no ser que ambos tipos de datos compartan la alineación en memoria.

Una plataforma de implementación determinada puede opcionalmente limitar dichas conversiones emitiendo errores de compilación o excepciones en tiempo de ejecución.

En cualquier caso las conversiones permitidas para los tipos punteros serán limitadas por el Módulo de Salida Zoe utilizado para implementar el programa de acuerdo a las clases de punteros que soporte dicho módulo de salida.

4 INSTRUCCIONES

Toda acción que se pueda realizar en el cuerpo de un método, como definir variables locales, llamar a métodos, asignaciones, etcétera son instrucciones.
Las instrucciones se agrupan formando bloques de instrucciones, que son listas de instrucciones encerradas entre llaves que se ejecutan en secuencia una tras otra. Es decir, la sintaxis que se sigue para definir un bloque de instrucciones es:

```
{
   <listaInstrucciones>
}
```

Toda variable que se defina dentro de un bloque de instrucciones sólo existirá dentro de dicho bloque. Tras él será inaccesible y podrá ser destruida por el recolector de basura. Estas variables reciben el nombre de variables locales. Para más información léase el capítulo 2.1 de este manual
Los bloques de instrucciones pueden anidarse, aunque si dentro de un bloque interno definimos una variable con el mismo nombre que otra definida en un bloque externo se considerará que se ha producido un error, ya que no se podrá determinar a cuál de las dos se estará haciendo referencia cada vez que se utilice su nombre en el bloque interno.

4.1 Variables Locales

Son variables que se definen en el cuerpo de los métodos y sólo son accesibles desde dicho cuerpo.
La sintaxis para definirlas es la siguiente:

```
<modificadores> <tipovariable> <nombrevariable>=<valor>,<nombrevariable>=<valor>, … ;
```

Para más información acerca de las variables locales léase el capítulo 2.1

```
// Ejemplo:
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   class Test{
   public:
      static void Main(string^[] args){
         string^ e = Console::ReadLine();
         for (int  i=0, suma=0;i < 20;i++){
            //utilización de la variable local  i dentro del ciclo for
            Console::WriteLine("el elemento es:" + i);
         }
      }
   }
}
```

En el ejemplo anterior se puede visualizar que la variable i es local al ciclo for, ya que cuando el mismo concluye su recorrido la variable i deja de existir.

4.2 Instrucciones Condicionales

Las Instrucciones Condicionales son instrucciones que permiten ejecutar ciertos bloques de instrucciones solo si se da determinada condición

4.2.1 Instrucción if

La Instrucción if permite ejecutar ciertas instrucciones sólo si se da una determinada condición. Su sintaxis es la siguiente:

```
if (<condición>)
{
      <Instrucciones>
}
else if (<condición>)
{
      <Instrucciones>
}
else if (<condición>)
{
      <Instrucciones>
}
else
{
      <Instrucciones>
}
```

Se evalúa la expresión condición, que ha de devolver un valor lógico. Si es cierta (devuelve true) se ejecutan el primer bloque de instrucciones y si es falsa (false) se ejecuta el bloque de las segundas instrucciones. La rama else es opcional y si se omite y la condición es falsa se ejecutan las instrucciones siguientes al bloque IF.

```
// Veamos un ejemplo:
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   class Test{
   public:
      static void Main(string^[] args){
         Console::WriteLine("Bien realizado");
         string^ e = Console::ReadLine();
         if (e == "")
         {
            Console::WriteLine("Debe ingresar algo");
            return ;
         }
         else
         {
            Console::WriteLine("Usted ingreso =" + e);
         }
      }
   }
}
```

4.2.2 Instrucción Switch

La sintaxis de esta Instrucción es la siguiente:

```
switch ( <expression> )
{
case <valor1>:
   <bloque1>
    <siguiente acción>
```

```
   break;
case <valor2>:
   <bloque2>
   <siguiente acción>
   break;
default:
   <bloque defautl>
   break;
}
```

El significado de esta instrucción es el siguiente: se evalúa <expresión>. Si su valor es <valor1> se ejecuta el <bloque1>; si es <valor2> se ejecuta el <bloque2>, y así para el resto de valores especificados. Si no es igual a ninguno de esos valores y se incluye la rama default, se ejecuta el bloque <default>, pero si no se incluye se pasa a ejecutar la instrucción siguiente al switch.

Los valores indicados en cada rama del switch han de ser expresiones constantes que produzcan valores de algún tipo básico entero, de una enumeración, de tipo char o de tipo string. Aunque todas las ramas son opcionales siempre se ha de incluir al menos una de ellas.

```
// Ejemplo:
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
using DotNET::System;

namespace switchess{
   class SwitchTest{
      static void Main(){
         Console::WriteLine("Elija el tamano de pizzas que desea: 1=Pequeña
         2=Mediana 3=Grande");
         Console::Write("Por favor seleccione un tamaño: ");
         string s = Console::ReadLine();
         int n = Convert::ToInt32(s);
         int precio = 0;
         switch (n){
            case 1:
               precio = 25;
               break ;
            case 2:
               precio = 30;
               break ;
            case 3:
               precio = 50;
               break ;
            default :
                     Console::WriteLine("Selecciónn invàlida. Por favor
               seleccione  1, 2, o 3");
               break ;
         }
                     Console::WriteLine("GRACIAS POR USAR NUESTROS SERVICIOS");
      }
   }
}
```

4.3 Instrucciones Iterativas

Son instrucciones que permiten ejecutar repetidas veces una instrucción o un bloque de instrucciones mientras se cumpla una condición. Es decir, permiten definir bucles donde ciertas instrucciones se ejecuten varias veces.

4.3.1 Instrucciones While

```
   while ( <condición> )
   {
        <instrucciones> ;
   }
```

Se evalúa la condición indicada (condición), que ha de producir un valor lógico. Si es cierta (valor lógico true) se ejecuta instrucciones y se repite el proceso de evaluación de condición y ejecución de las instrucciones hasta que deje de serlo. Cuando sea falsa (false) se pasará a ejecutar la instrucción siguiente al while. Cuando se habla de instrucciones puede ser una sola o un bloque de instrucciones
Dentro de las instrucciones de un while pueden utilizarse las siguientes dos instrucciones especiales:

- **break:** indica que se debe abordar la ejecución del bucle y continuarse ejecutando por la instrucción siguiente al While. Instrucción utilizada en el Switch
- **continue**: Indica que se ha de abordar la ejecución de las instrucciones y reevaluarse la condición del bucle, volviéndose a ejecutar las instrucciones si es cierta o pasándose a ejecutar la instrucción siguiente al while si es falsa Para más información consulte en Instrucciones de Salto

```
// Ejemplo de la Instrucción While:
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   class Test{
   public:
      static void Main(string^[] args){
                     Console::WriteLine("Debe ingresar algo");
                     //utilización del ciclo while
         while (e == ""){
            Console::WriteLine("Debe ingresar algo");
               Console::ReadLine();
         }
      }
   }
}
```

4.3.2 Instrucción do

La instrucción do while es una variante del while.
La sintaxis del mismo es:

```
do <instrucción> while (<condición>);
```

El do while primero ejecuta las instrucciones y luego pregunta si se cumple la condición, a diferencia del while que hace todo lo contrario, así el Do while ejecuta por lo menos una vez las instrucciones.
Do While está especialmente diseñado para los casos en los que haya que ejecutar las instrucciones al menos una vez aún cuando la condición sea falsa desde un principio

```
// Ejemplo de utilización del ciclo do while:
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;
```

```
namespace T{
   public class Test{
   public:
      static void Main(string^[] args){
         string^ e = "n";
         do{
            Console::WriteLine("Debe ingresar algo distinto de b");
            e = Console::ReadLine();
         } while (e != "b");
         Console::ReadLine();
         return ;
      }
   }
}
```

4.3.3 Instrucción for

La sintaxis:

```
for (<declaraciones>; <condición de repetición>; <expresiones de iteración>)
{
   instrucciones;
}
```

El significado de esta instrucción es el siguiente: se ejecutan las instrucciones de la primera parte (int i=0), que pueden usarse para definir e inicializar variables que luego se utilizarán en instrucciones. Luego se evalúa la condición (expresada en el segundo término i< length, que puede ponerse otra condición), y si es falsa se continúa ejecutando por la instrucción siguiente al ciclo for (es decir sale del ciclo y ejecuta la siguiente instrucción en línea de código); mientras que si es cierta se ejecutan las instrucciones indicadas, luego se ejecutan las instrucciones de modificación (representada por el tercer término de la expresión establecida en la sintaxis- i++) que como su nombre indica suele un usarse para modificar los valores de las variables que se usen en Instrucciones y luego se reevalúa la condición repitiendóse el proceso hasta que ésta última deje de ser cierta.
Al igual que con While , dentro de las instrucciones del for también pueden incluirse instrucciones continue y break, que puedan alterar el funcionamiento normal del bucle.

```
// Ejemplo:
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   class Test{
   public:
      static void Main(string^[] args){
         // ciclo for
         for (int  i=0;i < 20;i++)
         {
            Console::WriteLine("ciclo:" + i);
         }
      }
   }
}
```

4.3.4 Instrucción foreach

La sintaxis:

```
    foreach (<declaracion> in <expresión que retorna una matriz o colección>)
    {
       <instrucciones>
    }
```

El significado de esta instrucción es muy sencillo: se declara una variable para recorrer una matriz o una colección. En el cuerpo del mismo se ejecutan ciertas instrucciones, a gusto del programador.

A continuación se muestra un ejemplo en donde se puede observar como se utiliza esta instrucción.

Ejemplo de utilización del ciclo foreach:

```
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   class Test{
   public:
      static void Main(string^[] args){
         //declaración de Foreach
         foreach (string^ arg in args)
                        {
            Console::WriteLine(arg);
         }
      }
   }
}
```

4.4 Instrucciones de Salto

Las Instrucciones de salto pueden variar el orden normal en que se ejecutan las instrucciones de una programa, que consiste en ejecutarlas una tras otra en el mismo orden en que se hubiesen escrito en el código.

4.4.1 Instrucción Break

La instrucción Break solo puede incluirse dentro de los bloques de instrucciones asociadas a instrucciones iterativas o instrucciones switch e indica que se desea abordar la ejecución de las mismas y seguir ejecutando a partir de instrucción siguiente a ellas.

```
foreach (object^ var in collection_to_loop)
{
      if (b==1)break;
}

// Ejemplo de utilización de la instrucción break en un switch:

import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
using DotNET::System;

namespace switchess{
   class SwitchTest{
      static void Main(){
         Console::WriteLine("Elija el tamano de pizzas que desea: 1=Pequeña
         2=Mediana 3=Grande");
         Console::Write("Por favor seleccione un tamaño: ");
         string s = Console::ReadLine();
         int n = Convert::ToInt32(s);
         int precio = 0;
         switch (n){
            case 1:
```

```
            precio = 25;
            break ;
         case 2:
            precio = 30;
            break ;
         case 3:
            precio = 50;
            break ; //sale del switch
         default :
            Console::WriteLine("Selección inválida. Por favor
            seleccione  1, 2, o 3");
            break ;

      }
      Console::WriteLine("GRACIAS POR USAR NUESTROS SERVICIOS");
   }
  }
}
```

```
// Ejemplo de utilización de break con ciclo for:

import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
  class Test{
  public:
     static void Main(string^[] args){
        //declaración del ciclo for
        string^ [] nombres= new string^[2];
        nombres[0]= "pablo";
        nombres[1]= "joaquin";
        nombres[2]="pedro";
        for (int  i=0;i < 20;i++)
        {
           Console::WriteLine("el elemento vale:" + nombres[i]);
                          //cuando encuentra el nombre pedro sale del ciclo for
           if(nombres[i]=="pedro") break;
        }
     }
  }
}
```

4.4.2 Instrucción Continue

La Instrucción continue solo puede usarse dentro del bloque de instrucciones de una instrucción iterativa e indica que se desea pasar a reevaluar la condición de la misma sin ejecutar el resto de instruccones que contuviese. La evaluación de la condición se hará de la forma habitual; si es cierta se repite el bucle y si es falsa se continúa ejecutando por la instrucción que le sigue.

```
for (int i = 0; i < length; i++)
{
   if(b==1) continue;
}
```

Ejemplo de utilización de la instrucción continue con ciclo for:

```
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   class Test{
   public:
      static void Main(string^[] args){
         Console::WriteLine("Bien realizado");
         string^ e = Console::ReadLine();
         for (int  i=0;i < 20;i++){
            Console::WriteLine("i");
            if (i == 0){
               i = i + 1;
               // continue ejecuta el siguiente ciclo del for
               continue ;
            }
         }
      }
   }
}
```

En el ejemplo anterior cuando la variable i es igual a cero, se suma uno a esa variable y se procede a ejecutar el ciclo siguiente del for, en donde ahora i es igual 2.

4.4.3 Instrucción Return

Esta instrucción se usa para indicar cual es el objeto que tiene que devolver un método.

La sintaxis es:

```
return <ObjetoRetorno>;

ó

return;
```

La ejecución de esta instrucción provoca que se aborte la ejecución del método dentro del que aparece y que se devuelva <ObjetoRetorno> al método que lo llamó. Es obligatorio que todo método con tipo de retorno termine por un return.
Los métodos que devuelven void pueden tener un return con una sintaxis especial en la que no se indica ningún valor a devolver sino que simplemente se usa return para indicar que se desea terminar la ejecución del método. En este caso se indica solo con la palabra return y no se establece ningún objeto o valor para devolver.

Si se incluye un return dentro de un boque try con la cláusula finally, antes de devolverse el objeto especificado se ejecutarían las instrucciones de la claúsula finally. Si hubiesen varios bloques finally anidados, las instrucciones de cada uno se ejecutarían de manera ordenada, lo que no es posible es incluir un return dentro de la claúsula finally.

```
// Ejemplo de utilización de return:
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   class Test{
   public:
      static int Main(string^[] args) //devuelve un entero
              {
         Console::WriteLine("Bien realizado");
         string^ e = Console::ReadLine();
         for (int  i=0;i < 20;i++){
            Console::WriteLine("i");
                           //se devuelve el valor de la variable i
            if (i == 19){
                              return i;
            }
         }
                  //se devuelve cero
                  return 0;
      }
   }
```

4.4.4 Instrucción Nula

La instrucción nula es una instrucción que no realiza nada en absoluto. Su sintaxis consiste en escribir un simple punto y coma para representarla. O sea, es:

```
;
```

Suele usarse cuando se desea indicar explícitamente que no se desea ejecutar nada, lo que es útil para facilitar la legibilidad del código o, como veremos más adelante en el tema, porque otras instrucciones la necesitan para indicar cuándo en algunos de sus bloques de instrucciones componentes no se ha de realizar ninguna acción.

```
// Ejemplo de utilización de la instrucción nula:
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace test{
   class Publicidad{
   public:
      int numero;
      double precio;
      Publicidad(int a, double pre){
         numero = a;
         //utilización de la instrucción nula;
         if(a==0);
         precio = pre;
      }
   }
}
```

5 CLASES

Una clase es la definición de las características concretas de un determinado tipo de objeto.También se puede decir que una clase sirve como un contenedor y organizador para otras entidades definidas en su interior. Las clases son tipo con semántica de asignación por referencia. Eso significa que una variable o campo perteneciente al tipo almacena un puntero a una instancia del tipo pero nunca directamente una de esas instancias.
Las clases pueden contener estos tipos de declaraciones:

- ✓ **Campos:** los campos son todas las variables declaradas dentro de la clase. Los campos generalmente son privados
- ✓ **Métodos:** todo el código ejecutable en un lenguaje orientado a objetos debe estar encerrado en métodos. Los métodos son funciones definidas dentro de una clase
- ✓ **Propiedades:** Simulan la lectura del valor de un campo. Para más información Capítulo 5.8
- ✓ **Constructores:** parecen métodos, porque contienen código ejecutable pero solo pueden ser llamados a través de la instrucción New. Crean instancias de objetos a partir de clases. No es obligatorio definir un constructor para cada clase, y en caso de que no definamos ninguno el compilador creará uno por nosotros sin parámetros ni instrucciones.

Cuando un objeto ya no se necesita más, se destruye mediante una llamada al destructor de la clase. En Meta D++ la destrucción de los objetos suele hacerse de forma automatizada, es decir, a diferencia de lo que ocurre en otros entornos de programación, no es necesario destruir explícitamente un objeto para eliminarlo de la memoria, esa gestión de limpieza de objetos la realiza el recolector de basura, el cual decide cuando un objeto no se necesita más y en ese caso lo elimina dejando libre la memoria. Sin embargo el funcionamiento especifico dependerá del Modulo de Salida Zoe utilizado para la construcción del programa. Adicionalmente los Módulos de Salida Zoe clases A y B soportan almacenamiento de montón (heap) no manejado, en dicho caso debe liberarse la memoria usando la expresión delete. Estos casos especiales no son tratados en el presente manual por tratarse de un manual sobre conceptos más básicos.

```
// Ejemplo de los conceptos mencionados arriba
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T
{
   class Perro
   {
   private:
      int edad;   //campos
      string^ nombre; //campos
   public:
      Perro (int edad) //constructor con paramétros
      {
         this.edad = edad;
         nombre = "blackie";
      }
      int property Edad{ //propiedad
         get { return edad; }
         set { edad = value; }
      }
   }
}
```

5.1 Definición de una Clase

Sintaxis de definición de clases

```
   <modificadores> class <nombredeclase> inherits <clase base> implents <interfaces>
   {
       <miembros>
   }
```

```
Ejemplo:
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
       //creación de la clase alumno
       class Alumno inherits Persona implements IEntity
       {
      int edad;
      string^ nombre;
   public:
      Alumno(int edad){
         this.edad = edad;
         nombre = "jose";
      }
   }
}
```

5.2 Creación de Objetos

5.2.1 Operador New

La sintaxis para crear un objeto es el siguiente:

```
new <nombretipo> (<argumentos>)
```

Este operador crea un nuevo objeto del tipo cuyo nombre se le indica y llama durante su proceso de creación al constructor del mismo apropiado según los valores que se le pasen en <parámetros>, devolviendo una referencia al objeto recien creado . El operador new devuelve una referencia a la dirección de memoria en donde se ha creado
Un objeto puede tener múltiples constructores, aunque para diferenciar unos de otros es obligatorio que se diferencien en el número u orden de los parámetros que aceptan.

```
Persona^ nuevaPersona = new Persona("jose", 12);
int[] matriz = new int[] = {1, 2, 3}
```

5.3 Modificadores de Acceso

La encapsulación se logra a través de los modificadores de acceso. Estos modificadores de acceso son modificadores que se colocan delante de los miembros (seguido de dos puntos a la forma de C++) y tipos de datos para indicar desde donde se puede accederse a ellos, entendiéndose por acceder el hecho de usar su nombre.
Por defecto se considera que los miembros de un tipo de datos solo son accesibles desde código situado dentro de la definición del mismo, aunque esto puede cambiarse precediéndolos de uno de los siguientes modificadores.

5.3.1 Public:

Puede ser accedido desde cualquier parte del código.

```
Ejemplo:
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   class Profesor{
      int legajo;
      string^ apellido;
        //utilización del modificador de acceso público
   public:
       Profesor(string^ apellidos){
         legajo = 222;
         apellido = apellidos;
      }
   }
   class Test{
   public:
      static void Main(string^[] args){
         Profesor^ a = new Profesor("Perez");
         Console::WriteLine("el constructor de profesor es público para cualquier
instancia de profesor");
      }
   }
}
```

5.3.2 Protected:

Desde una clase sólo puede accederse a miembros protected de objetos de esa misma clase o subclases suyas.

```
// Ejemplo:

import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   class A{
   // Se declaro el campo x como protegido
   protected:
      int x;
   public:
      static void F(A^ a, B^ b){

         a.x = 1;
         b.x = 1;
      }
   }
   class B inherits A{
   public:
      static void F(A^ a, B^ b){
         a.x = 2;
         b.x = 3;
      }
   }
   public class Test{
   public:
      static void Main(string^[] args){
         B^ b = new B();
         Console::WriteLine(B.x); // error no puede acceder a x
```

```
      }
   }
}
```

5.3.3 Private:

Sólo puede ser accedido desde el código de la clase a la que pertenece. Es el considerado por defecto.

```
// Ejemplo:
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   class Cliente{
      string^ name;
      int age;
   }
   class Proveedor inherits Cliente{
      void foo(){
         // error no se puede acceder a metodo privado heredado
         this.age = 12;
      }
   }
}
```

5.4 Modificadores de Acceso Especial

Para las clases ClassFactory se utilizan tres modificadores de acceso especial, los cuales son: **iprotected, ipublic e iprivate.** Todos ellos están disponibles únicamente para la utilización de código para ClassFactorys.

5.5 Utilización del Puntero this

La palabra clave **this** hace referencia a la instancia actual de la clase. Dentro del código de cualquier método de instancia siempre es posible hacer referencia al propio objeto usando la palabra reservada `this`. El puntero this se define en Meta D++ como una referencia al tipo de la clase, por lo cual a diferencia de C++ no se debe utilizar el operador de acceso a miembro de puntero "->" como en C++ sino simplemente el "." (acceso a miebro).

```
// Ejemplo:
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   class Profesor{
      int legajo;
      string^ apellido;
   public:
       Profesor(string^ apellido){
                    //utilización del puntero this
         this.legajo = 222;
         this.apellido = apellido;
      }
   }
}
```

En el ejemplo anterior se puede observar que se utilizo el puntero this para asignar el apellido que se pasa como parámetro al constructor de la clase Profesor para almacenarlo en el campo

que lleva el mismo nombre que el parámetro (apellido). En dicho caso si no se utilizara this para acceder al campo no se produciría el resultado deseado ya que se asignaría el valor del parámetro al mismo parámetro en lugar de asignar el valor del parámetro al campo.

5.6 Tipos de Clases

5.6.1 Clases Abstractas

Una Clase Abstracta es aquella que forzosamente se ha de derivar si se desea que se puedan crear objetos de la misma o acceder a sus miembros estáticos.
Para definir una clase abstracta se antepone `abstract` a su definición.

```
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   abstract class A{
   protected:
      int x;
   public:
      void foo(){
         this.x = 1;
      }
      abstract void foo2(int value);
   }
   class B inherits A{
   public:
      void foo2(int value){
         this.x = value;
      }
      int property X{
         get{ return this.x; }
      }
   }
   class Test{
   public:
      static void Main(string^[] args){
         A^ a = new A(); // error no se puede crear instancia
         A^ a2 = new B(); // correcto
         B^ b = new B();
         b.foo();
         Console::WriteLine(b.X);
         b.foo2(12);
         Console::WriteLine(b.X);
      }
   }
}
```

La clase A del ejemplo anterior es abstracta y por tanto no pueden crearse instancias de la misma, sin embargo si pueden declarase variables de dicho tipo.
Si una clase declara al menos un miembro de tipo abstracto (con el modificador `abstract`) entonces la clase debe marcarse como abstracta.
No se puede marcar a un miembro como abstracto y virtual al mismo tiempo, esto es porque simplemente si un miembro es abstracto también es virtual.

5.6.2 Clases Finales

Una clase final es una clase que no puede tener clases hijas y para definirlas basta con anteponer el modificador de acceso `final` a la definición de una clase normal.
La sintaxis es la siguiente:

```
  final  class <nombre de clase> { }
```

Una utilidad de definir una clase final es que permite que las llamadas a sus métodos virtuales heredados se realicen tan eficientemente como si fuesen no virtuales, pues al no poder existir clases hijas que los redefinan no puede haber polimorfismo. No se permite definir miembros virtuales dentro de este tipo de clases, ya que al no poder heredarse de ellas es algo sin sentido en tanto que nunca podrían redefinirse.
Cuando una clase es final se reduce enormemente su capacidad de reutilización mediante herencia.
La principal causa de incluir este tipo de clases es que permiten asegurar que ciertas clases críticas nunca podrán tener clases hijas y sus variables siempre almacenarán objetos del mismo tipo.
Tengáse en cuenta que es absurdo definir simultaneamente una clase como abstracta y final, pues nunca podrá accederse a la misma al no poderse crear clases hijas suyas que definan sus métodos abstractos.

5.7 Métodos y Propiedades

5.7.1 Concepto de Método y Propiedades

Un método es un conjunto de instrucciones a las que se da un determinado nombre de tal manera que sea posible ejecutarlas en cualquier momento sin tener que reescribir sino usando sólo el nombre. A estas instrucciones se les denomina cuerpo del método y a su ejecución a través de su nombre se le denomina llamada al método.
La ejecución de las instrucciones de un método puede producir como resultado un objeto de cualquier tipo. A este objeto que devuelve se le llama valor de retorno del método y es opcional, pudiéndose escribir métodos que no devuelvan ninguno.
La ejecución de las instrucciones de un método puede depender del valor de unas variables especiales llamadas párametros del método, de manera que en función del valor que se le dé a estas variables en cada llamada la ejecución del método se pueda realizar de una u otra forma y podrá producir uno u otro valor de retorno.
Al conjunto formado por el nombre del método y el número y el tipo de parámetros se le conoce como *Firma del Método*. La *firma* del método es lo que verdaderamente identifica al mismo, de modo que es posible definir en un mismo tipo varios métodos con idéntico nombre siempre y cuando tengan distintos parámetros. Cuando esto ocurre se dice que el método que tiene ese nombre es sobrecargado.

Para definir un método hay que indicar cuáles son las instrucciones que forman su cuerpo como cuál es el nombre que se le dará, cual es el tipo de objeto que puede devolver y cuáles son los parámetros que puede tomar. Esto se indica definiendólo así:

```
<modificadoracceso><tiporetorno> <nombremetodo>(<parámetros>)
{
}
```

En <tiporetorno> se indica cuál es el tipo de datos que el método devuelve y si no devuelve ninguno se ha de escribir void en su lugar.
Como nombre del método se puede poner en <nombremétodo> cualquier identificador válido.
Aunque es posible escribir métodos que no tomen parámetros, si un método los toma se ha de indicar en <parámetros> cuál es el nombre y tipo de cada uno, separándolos con comas si son más de uno.
El cuerpo del método también es opcional, pero si el método retorna algún tipo de objeto entonces ha de incluir al menos una instrucción return que indique cuál es el objeto.

```
// Ejemplo de Método

import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
using DotNET::System;

namespace test{
   class Alumno{
      float nota1;
      float nota2;
      int legajo;
      string^ apellido;
      float promedios;
   public:
       Alumno(int lega, string^ apellidos){
         legajo = lega;
         apellido = apellidos;
      }
              //creación de método
      float promedio(int n1, int n2){

         promedios = n1 / n2;
         return promedios;
      }
   }
   class App{
   public:
      static void Main(string^[] args){
         Alumno^ a = new Alumno(1,"Vila");
         float p = a.promedio(7, 6);
         Console::WriteLine("El promedio del alumno es:" + p);
      }
   }
}
```

Una propiedad es una mezcla entre el concepto de campo y el concepto de método . Una propiedad no almacena datos, sino sólo se utiliza como si los almacenase.
Para definir una propiedad se usa la siguiente sintaxis:

```
     <tipopropiedad>  property <nombrepropiedad>
      { set
        {<código de escritura>}
       get
        {<código de lectura>}
      }
```

Una propiedad así definida sería accedida como si de un campo de tipo <tipopropiedad> se tratase pero en cada lectura de su valor se ejecutaría el coódigo de lectura y en cada escritura de un valor en ella se ejecutaría el código de escritura.
Al escribir los bloques de código set y get hay que tener en cuenta que dentro del código set se puede hacer referencia al valor que se solicita asignar a través de un parámetro especial

del mismo tipo de dato que la propiedad llamado value y dentro del código get se ha de devolver siempre un objeto del tipo de dato de la propiedad.
El orden en que aparezcan los bloques get y set es irrelevante.
Es posible definir propiedades que solo tengan el bloque get (propiedades de sólo lectura) o que solo tengan el bloque set (propiedades de sólo lectura). Lo que no es válido es definir propiedades que no incluyan ninguno de los dos bloques.

```
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
using DotNET::System;

namespace test{
   class Alumno{
      float nota1;
      float nota2;
      int legajo;
      string^ apellido;
      float promedios;
   public:
      Alumno(int lega, string^ apellidos){
         legajo = lega;
         apellido = apellidos;
      }
            //creación de método
      float promedio(int n1, int n2){

         promedios = n1 / n2;
         return promedios;
      }
   }
   class App{
   public:
      static void Main(string^[] args){
         Alumno^ a = new Alumno(1,"Vila");
         float p = a.promedio(7, 6);
         Console::WriteLine("El promedio del alumno es:" + p);
      }
   }
}
```

5.7.2 Acceso a Propiedades

La forma de acceder a una propiedad, ya sea para lectura o escritura, es la misma que la que se usaría para acceder a un campo de su mismo tipo.
Por ejemplo se podría acceder a la propiedad de un objeto de la clase B definida anteriormente de la siguiente manera:

```
B obj= new B();
obj.PropiedadEjemplo++;
```

5.7.3 Llamada a Métodos

La forma en que se puede llamar a un método es la siguiente:

```
<tipo> <nombremétodo> (<valoresparámetro>)
```

Ahora en <tipo> ha de indicarse el tipo donde se ha definido el método o algún subtipo suyo. Sin embargo, si el método pertenece al mismo tipo que el código que lo llama entonces se puede usar la siguiente expresión abreviada:

```
<nombreMétodo>(<valoresParámetros>)
```

```
Ejemplo:
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
using DotNET::System;

namespace test{
   class Alumno{
      float nota1;
      float nota2;
      int legajo;
      string^ apellido;
      float promedios;
   public:
       Alumno(int lega, string^ apellidos){
         legajo = lega;
         apellido = apellidos;
      }
            //creación de método
      float promedio(int n1, int n2){

         promedios = n1 / n2;
         return promedios;
      }
   }
   class App{
   public:
      static void Main(string^[] args){
         Alumno^ a = new Alumno(1,"Vila");
         float p = a.promedio(7, 6);
         Console::WriteLine("El promedio del alumno es:" + p);
      }
   }
}
```

5.7.4 Parámetros

Los parámetros se inicializan en cada llamada al método al que pertenecen con los valores especificados al llamarlo.
La forma en que se define cada parámetro de un método depende del tipo de parámetro que se trate. Se puede hablar de cuatro tipos de parámetros: parámetros de salida, parámetros de entrada, parámetros por referencia y parámetros de número indefinido.

5.7.4.1 Parámetros de Entrada

Un parámetro de entrada recibe una copia del valor que almacenaría una variable del tipo del objeto que se le pase. Por lo tanto, si el objeto es un tipo de valor se le pasará una copia del objeto y cualquier modificación que se haga al parámetro dentro del cuerpo del método no afectará al objeto original sino a su copia, mientras que si el objeto es de un tipo de referencia entonces se le pasará una copia de la referencia al mismo y cualquier modificación que se haga al parámetro dentro del método también afectará al objeto original ya que en realidad el parámetro referencia a ese mismo objeto original.
Para definir un método de entrada basta indicar cuál es el nombre que se le desea dar y cuál es el tipo de datos que podrá almacenar. Para ello es la siguiente sintaxis:

```
 <tipoparámetro> <nombreparámetro>
```

Se usa la instrucción return para indicar cuál es el valor que ha de devolver el método. Este es el resultado de ejecutar la expresión part1 + part2, es decir la suma de los valores pasados a sus parámetros part1 y part2 y devuelve un int con la suma de ambos

```
// Ejemplo:
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   class ParametroEntradaejemplo{
   public:
      int b;
      int suma(int parte1, int parte2){
         return parte1 + parte2;
      }
   }
   class Test{
   public:
      static void Main(string^[] args){
         ParametroEntradaejemplo p = new ParametroEntradaejemplo();
         p.suma(12, 10);
      }
   }
}
```

5.7.4.2 Parámetros de Salida

Un parámetro de salida se diferencia de uno de entrada en que todo cambio que se le realice en el código del método al que pertenece afectará al objeto que se le pase al llamar dicho método tanto si éste es de un tipo valor o de un tipo referencia. Esto se debe a que lo que estos parámetros se les pasa es siempre una referencia al valor que almacenaría una variable de tipo del objeto que se le pase.
Cualquier parámetro de salida de un método siempre ha de modificarse dentro del cuerpo del método.
Este tipo de parámetros permiten diseñar métodos que devuelvan múltiples objetos : un objeto se devolvería como valor de retorno y los demás se devolverían escribiéndolos en los parámetros de salida.
La sintaxis es la siguiente:

```
     out <tipoparametro> <nombreparametro>
```

```
// Ejemplo:
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   class ParametroSalida{
   public:
      void F(out int* b){
         *b = 1;
      }
   }
   class Test{
   public:
      static void Main(string^[] args){
         ParametroSalida^ p = new ParametroSalida();
         int val=5;
```

```
        p.F(&val);
      }
   }
}
```

5.7.4.3 Parámetros de número indefinido

Se permite diseñar métodos que puedan tomar cualquier número de parámetros. Para ello hay que indicar como último parámetro del método un parámetro de algún tipo de una tabla unidimensional.
La sintaxis es la siguiente:

```
    static void f(int x, params object[] extras)
    {
    }
```

Todos los parámetros que número indefinido que se pasen al método al llamarlo han de ser del mismo tipo que de la matriz.

```
// Ejemplo:
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   class ParametroEntrada{
   public:
      int b;
      void F(int x, params int[] extras){
         b = x;
      }
   }
   class Test{
   public:
      static void Main(string^[] args){
         ParametroEntrada^ var = new ParametroEntrada();
         //Forma implicita
         var.F(4, 1, 2, 3);
         //Forma explicita usando una matriz
         var.F(4, new int[] = {1,2,3});
      }
   }
}
```

En el ejemplo anterior se visualiza que se establece la cantidad que el usuario desee al Método F().

5.7.4.4 Parámetros por Referencia

Un parámetro por referencia es similar a un parámetro de salida sólo que no es obligatorio modificarlo dentro del método al que pertenece, por lo que será obligatorio pasarle una variable inicializada ya que no se garantiza su inicialización en el método.
Los parámetros por referencia se definen igual que los parámetros de salida pero sustituyendo el modificador out por el modificador ref. Del mismo modo, al pasar valores a parámetros por referencia también hay que precederlos del ref.

```
// Ejemplo:
import "Microsoft", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
using DotNET::System;
using DotNET::System::IO;
```

```
using DotNET::System::Collections;

namespace Sample{
   public class Test{
   public:
      string^ Name;
      int Value;
      override string^ ToString(){
         return "Name : "+ Name+ " , Value: "+Value.ToString();
      }
   }
   public class App{
   public:
      static int Main(string^[] args){
         Test^ myVar = new Test();
         myVar.Name = "Juan";
         myVar.Value = 5;
         Test^ myVar2 = new Test();
         myVar2.Name = "Pedro";
         myVar2.Value = 15;

         Console::WriteLine(myVar.ToString());
         Console::WriteLine(myVar2.ToString());

         TestMethod(&myVar, &myVar2);

         Console::WriteLine(myVar.ToString());
         Console::WriteLine(myVar2.ToString());

         Console::ReadLine();
         return 0;
      }
      static void TestMethod(inout Test*^ testParam, out Test*^ testParam2){
         testParam.Name = "Belen";
         testParam.Value = 12;

         testParam2 = new Test();
         testParam2.Name = "Daniela";
         testParam2.Value = 20;
      }
   }
}
```

5.7.5 Sobrecarga de Métodos

La sobrecarga de métodos consiste en declarar varios métodos con el mismo nombre en la misma clase, pero siempre que su lista de argumentos sea distinta. Se distinguen examinando la lista de los parámetros. No se pueden declarar dos métodos que tengan el mismo nombre, los mismos tipos y nombres de parámetros, y que se distingan únicamente de los tipos de datos que devuelven, así podemos ver esto a través de un ejemplo.

```
// Ejemplo:
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   class EjemploSobrecarga{
   public:
      int suma(int a, int b){
         return a + b;
      }
      float suma(int a, int b){ //ERROR.
         return a + b;
      }
   }
```

```
   class Test{
   public:
      static void Main(string^[] args){
         Console::WriteLine("Bien realizado");
         int n;
         EjemploSobrecarga^ b = new EjemploSobrecarga();
                    //llamada al primer método sobrecargado
         n = b.suma(2, 3);
                    //llamada al segundo método sobrecargado
                     n= b.suma(2, 3);
      }
   }
}
```

En el ejemplo anterior se puede observar que hay dos métodos que se llaman igual, con los mismos tipos y nombres de parámetros, y lo único que lo diferencia es el tipo de dato que devuelve, ya que el primero devuelve un entero (int) mientras que el segundo devuelve un float, por lo que el compilador dará error.
A continuación se muestra un ejemplo de cómo sería correcto definir la sobrecarga.

```
// Ejemplo:
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   class EjemploSobrecarga{
   public:
      int suma(int a, int b){
         return a + b;
      }
      int  suma(int a, int b, int c){ // método sobrecargado
         return a + b + c;
      }
   }
   class Test{
   public:
      static void Main(string^[] args){
         Console::WriteLine("Bien realizado");
         int n;
         EjemploSobrecarga^ b = new EjemploSobrecarga();
         n = b.suma(2, 3);  //llamada al primer método sobrecargado
                     n= b.suma(2,3,4);  // llamada al segundo método sobrecargado
      }
   }
}
```

En el ejemplo anterior se observa que el método suma está sobrecargado, ya que hay dos métodos con ese mismo nombre pero lo que cambia es la cantidad de parámetros que usan como argumentos

5.7.6 Sobrecarga de Operadores

Una de las características que proporciona el lenguaje Meta D++ es la capacidad de sobrecarga de operadores. De modo análogo a la sobrecarga de funciones, la sobrecarga de operadores permite al programador dar nuevos significados a los símbolos de los operadores existentes en Meta D++.
La sobrecarga de operadores permite redefinir ciertos operadores, como "+" y "-", para usarlos con las clases definidas por el programador.

Se llama sobrecarga de operadores porque estamos reutilizando el mismo operador con un número de usos diferentes, y el compilador decide cómo usar ese operador dependiendo sobre qué opera.
Los operadores lógicos && y || pueden ser sobrecargados para las clases definidas por el programador.
Para sobrecargar un operador, debemos definir una función de sobrecarga del operador en la clase que nos interesa o en el módulo o procedimiento.
Si queremos sumar dos tipos de datos enteros usamos el operador (+), si queremos comparar los valores de variables del mismo tipo usamos los operadores de comparación (>, <, ó =), y cuando compilamos no nos da error ya que el compilador sabe cómo realizar esas operaciones porque se ha definido código para tratar las mismas. El problema ocurre cuando queremos usar esos operadores con tipos definidos por programadores. A continuación se explica esto a través de un ejemplo:
A continuación se muestra un ejemplo:

```
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   public class Coordenadas{
   public:
      int x;
      int y;
       Coordenadas(int l, int w){
         x = l;
         y = w;
      }
   }
   public class Test{
   public:
      //error ya que el compilador no sabe como sumar;
      static void Main(string^[] args){
         Coordenadas^ c1 = new Coordenadas(2,3);
         Coordenadas^ c2 = new Coordenadas(3,3);
         Coordenadas^ c3 = c1 + c2;
      }
   }
}
```

En el ejemplo anterior el compilador en el momento de la ejecución lanzará error debido a que no puede utilizar el operador (+) con los tipos de datos definidos por el programador. Para poder solucionar este problema, se recurre a crear un método (Add) a través del cual se le indicará al compilador como sumar esos tipos de datos. A continuación se muestra el código correspondiente:

```
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   public class Coordenadas{
   public:
      int x;
      int y;
       Coordenadas(int l, int w){
         x = l;
         y = w;
      }
```

```
      static Coordenadas^ Add(Coordenadas^ a, Coordenadas^ b){
         return new Coordenadas(a.x + b.x,a.y + b.y);
      }
      override string^ ToString(){
         return x.ToString() + "," * y.ToString();
      }
   }
   public class Test{
   public:
      static void Main(string^[] args){
         Coordenadas^ c1 = new Coordenadas(2,3);
         Coordenadas^ c2 = new Coordenadas(3,3);
         Coordenadas^ c3 = Coordenadas.Add(c1, c2);
         Console::WriteLine("El valor de la Coordenada 3 es:" + c3.ToString());
      }
   }
}
```

5.7.6.1 Sobrecarga de Operadores unarios y binarios

Los operadores unitarios son aquellos que requieren un operando. Ejemplo de ellos son:

- ✓ Unarios: Incremento (++), Decremento (--), Negación (!)
- ✓ Binarios: Aritméticos (-, +, *, / , etc.), de Comparación (==, !=, >, <)

La sintaxis es la siguiente:

```
<tipo> operador <operador unitario>();
```

En donde <tipo> es la clase para la que estamos sobrecargando el operador

```
// Ejemplo de sobrecarga de  Operador Incremento(++):
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   public class Tiempo{
   public:
      int minutos;
      int horas;
      Tiempo(int horas, int minutos){
         this.minutos = minutos;
         this.horas = horas;
      }
      static Tiempo^ operator ++(Tiempo^ h){
         h.minutos++;
         while (h.minutos >= 60){
            h.minutos = 0;
            h.horas++;
         }
         return h;
      }
   }
   class Test{
   public:
      static void Main(string^[] args){
         Tiempo^ n = new Tiempo(12,59);
         n++;
         Console::WriteLine(n.minutos);
         Console::WriteLine(n.horas);
      }
   }
}
```

```
// Ejemplo de sobrecarga del Operador de Multiplicación(*)
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   public class Coordenadas{
   public:
      int x;
      int y;
       Coordenadas(int l, int w){
         x = l;
         y = w;
      }
      static Coordenadas^ operator *(Coordenadas^ a, Coordenadas^ b){
         return new Coordenadas(a.x*b.x,a.y*b.y);
      }
   }
   public class Test{
   public:
      static void Main(string^[] args){
         Coordenadas^ c1 = new Coordenadas(2,3);
         Coordenadas^ c2 = new Coordenadas(3,3);
         Coordenadas^ c3 = new Coordenadas(4,3);
         c3*;
         Console::WriteLine("El valor de la Coordenada 3 es:" + c3.ToString());
      }
   }
}

// Ejemplo de sobrecarga del Operador de Decremento (--)
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   public class Resta{
   public:
      int nro1;
      int nro2;
      Resta(int nro1, int nro2){
         this.nro1= = nro1;
         this.nro2 = nro2;
      }
      static Resta^ operator --(Resta^ h){
          if(h.nro1> h.nro2) h.nro1 - h.nro2;
          else h.nro2 - h.nro1;
         return h;
      }
   }
   class Test{
   public:
      static void Main(string^[] args){
         Resta^ n = new Resta (12,59);
         n--;
         Console::WriteLine(n.nro1);
         Console::WriteLine(n.nro2);
      }
   }
}
```

5.7.6.2 Sobrecarga de Operadores de Comparación

Otros operadores que podemos sobrecargar son los operadores que se utilizan para realizar comparaciones. En estos casos las sobrecargas deben hacer de a pares, es decir si se sobrecarga el operador (<) también hay que sobrecargar el operador (>). El caso del operador de igualdad (==) y de desigualdad (! =) siguen esta misma regla.

```
// Ejemplo de sobrecarga de los operadores iguales y distintos
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   public class Complejo{
   public:
      float partereal;
      float parteimaginaria;
      Complejo(float nro1, float nro2){
         parteimaginaria = nro2;
      }
      static bool operator ==(Complejo^ h1, Complejo^ h2){
         if (h1.partereal == h2.partereal && h1.parteimaginaria == h2.parteimaginaria){
            return true;
         }
         return false;
      }
      static bool operator !=(Complejo^ h1, Complejo^ h2){
         if (h1.partereal != h2.partereal && h1.parteimaginaria != h2.parteimaginaria){
            return true;
         }
         return false;
      }
   }
   class Test{
   public:
      static void Main(string^[] args){
         Complejo^ n = new Complejo(2,6);
         Complejo^ m = new Complejo(2,7);
         Console::WriteLine(n==m);
         Console::WriteLine(n!=m);
      }
   }
}
```

5.7.6.3 Operadores que no se sobrecargan

Algunos operadores no se pueden sobrecargar, como por ejemplo operador de asignación (=) ni otros compuestos como la asignación y la suma (+=).

5.7.7 Miembros de Clase

Una clase puede poseer miembros o campos denominados **estáticos** y miembros denominados de **instancia**. Los miembros estáticos son aquellos declarados con el modificador "`static`", los miembros de instancia son los que no utilizan éste modificador en su declaración. En general se refiere a los miembros estáticos como pertenecientes a la clase y los miembros de instancia pertenecientes a los objetos de esa clase.

Los **campos estáticos** sólo se inicializan la primera vez que se accede al tipo al que pertenecen, pero no en sucesivos accesos. Estos accesos pueden ser tanto para crear objetos de dicho tipo como para acceder a sus miembros estáticos. La inicialización se hace de modo que en primer lugar se dé a cada variable el valor por defecto correspondiente a su tipo, luego se dé a cada una el valor inicial especificado al definirlas, y por último se llame al constructor del tipo. Un constructor de tipo es similar a un constructor normal sólo que en su código únicamente puede accederse a miembros **static**.

Los **campos no estáticos** se inicializan cada vez que se crea un objeto del tipo de dato al que pertenecen. La inicialización se hace del mismo modo que en el caso de los campos estáticos, y

una vez terminada se pasa a ejecutar el código del constructor especificado al crear el objeto. En caso de que la creación del objeto sea el primer acceso que se haga al tipo de dato del mismo, entonces primero se inicializarán los campos estáticos y luego los no estáticos.
Para acceder a un miembro del objeto se utiliza la siguiente sintaxis:

```
<Identificador del objeto>.<Miembro>
```

Para acceder a un miembro estático es necesario utilizar la siguiente sintaxis:

```
<clase>::<campo>
```

Esto se puede ver en el ejemplo anterior, cuando se accede al campo estático intereses, en donde se establece el nombre de la clase y luego el nombre del campo (intereses):

```
Ejemplo de Miembros Estáticos:
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   class Cuenta{
      // Campos privados para almacenar los datos
      double _tiempoDeposito;
      double _saldo;
               //miembro estático
      static double _intereses;
   public:
      static double property Intereses {
         get {
            return _intereses;
         }
         set {
            _intereses = value;
         }
      }
       Cuenta(double saldo, double tiempoDeposito){
         // Constructor de la clase
         _saldo = saldo;
         _tiempoDeposito = tiempoDeposito;
      }
      double SaldoActual(){
         return _saldo * _tiempoDeposito * _intereses / 100;
      }
   }
   class Test{
   public:
      static void Main(string^[] args){
         Cuenta^ c = new Cuenta(1200,12);
         // Accedo a un miembro estatico
         Console::WriteLine("Cuenta::Intereses:"+     Cuenta::Intereses.ToString());
         // Accedo a un miembro de instancia
         Console::WriteLine("Saldo actual : " + c.SaldoActual());
      }
   }
}
```

5.7.8 Métodos Virtuales, Métodos Override y Métodos Abstractos

5.7.8.1 Métodos Virtuales

Los métodos virtuales son útiles cuando es necesario dar una nueva definición al método en las clases hijas.

El método debe ir precedido de la palabra reservada Virtual en la clase padre y en la clase hija debe definirse el método como Override .Si se precede de Override un método en la clase hija y el método de la clase padre no va precedido de Virtual se produce un error de compilación.
No se puede definir un método como virtual y override a la vez. Tampoco se pueden declarar como estáticos ni privados. Los métodos virtuales son polimórficos.
Se puede sustituir un método override. Un método override es virtual de manera implícita por lo que no se puede declarar explícitamente virtual. No se puede declarar un método override como estático o privado.
Para definición de estos métodos es necesaria la siguiente sintaxis:

```
<public> virtual <nombremétodo>
```

```
// Ejemplo:
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   abstract class C{
   protected:
      string^ _nombre;
      int _edad;
   public:
      C(string^ nombre){
         _nombre = nombre;
      }
      //se definio el metodo abstracto Edad, pero no se implemento
      virtual void Edad(int edad){
      }
   }
   class B inherits C{
   public:
      B(string^ nombre, int edad):base(nombre) {
         //constructor de B
         _edad = edad;
      }
      override void Edad(int edad){
         //se implemento el metodo Edad definido en la clase derivada
         _edad = edad;
      }
   }
   class Test{
   public:
      static void Main(string^[] args){
         B^ b = new B("Belen",2);
         b.Edad(22);
         Console::WriteLine("Se modifico la edad.");
      }
   }
}
```

5.7.8.2 Métodos Abstractos

Los métodos abstractos son aquellos que no se da una implementación directa en la clase que lo define sino que se los implementa en las clases hijas. No es necesario que todos los métodos dentro de la clase sean abstractos pero al menos uno de ellos debe serlo. Para los métodos abstractos se debe anteponer el modificar abstract y sustituir el código del cuerpo por “;”.
Todo método abstracto es implícitamente virtual.

```
// Ejemplo:
```

```
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   abstract class C{
   protected:
      string^ _nombre;
      int _edad;
   public:
      C(string^ nombre){
         _nombre = nombre;
      }
      //se definio el metodo abstracto Edad, pero no se implemento
      abstract void Edad(int edad);
   }
   class B inherits C{
   public:
      B(string^ nombre, int edad):base(nombre) {
         //constructor de B
         _edad = edad;
      }
      override void Edad(int edad){
         //se implemento el metodo Edad definido en la clase derivada
         _edad = edad;
      }
   }
   class Test{
   public:
      static void Main(string^[] args){
         B^ b = new B("Belen",2);
         b.Edad(22);
         Console::WriteLine("Se modifico la edad.");
      }
   }
}
```

5.8 Espacio de Nombres e Importación de Tipos

5.8.1 Definición de Espacio de Nombre

Los espacios de nombres son el conjunto de clases, funciones y tipos de datos que una aplicación puede utilizar. Además puede contener otros espacios de nombres. Un espacio de nombres es simplemente un contenedor lógico que puede albergar otros espacios de nombres anidados o tipos de datos.

Para indicar que se utiliza un determinado namespace se utiliza la palabra reservada Using.

La sintaxis para la declaración de un espacio de nombres es la mostrada a continuación:

```
Namespace_Decl:
      namespace Complete_Class_Name { Namespace_Members }
```

Un espacio de nombre se declara con la palabra clave "`namespace`" seguida de un nombre calificado, el cual puede ser un nombre simple y el cuerpo del espacio de nombres entre llaves.

El nombre simple del nombre calificado en la declaración de nombres es el nombre del espacio de nombres declarado. La declaración de un espacio de nombre crea un nuevo espacio de declaraciones, los tipos y espacios de nombres miembros del espacio de nombre introducirán nombres dentro de éste espacio de declaraciones.

Un espacio de nombre declarado con un nombre calificado o declarado dentro de otro espacio de nombres se dice que es un espacio de nombres anidado.

No es posible asignar un nivel de acceso a un espacio de nombres, estos son siempre públicos y pueden ser accedidos desde cualquier parte del programa, sin embargo los tipos declarados dentro de los espacios de nombres pueden ser privados evitando su utilización desde otros espacios de nombres.

```
// Ejemplos de declaración de espacio de nombre
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
using DotNET::System;

namespace Test{
   namespace N2{
      class A{
         static void Main(string^[] args){
            B^ varB;
            Console::WriteLine("Esto es una prueba");
         }
      }
   }
   class C{
   public:
      void test(){
         Test::N2::A^ varA;
      }
   }
}

namespace Test::N2{
   class B{
   public:
      void foo(){
         A^ varA;
         Test::C^ varC;
      }
   }
}
```

Siempre que querramos hacer un programa para .NET deberemos importar el assembly por defecto de .NET, es decir “mscorlib”, esto se hace con la directiva:

```
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
```

Luego normalmente agregaremos las directivas “using”:

```
using DotNET::System;
using DotNET::System::Collections;
```

Si se requiere utilizar un tipo importado desde .NET simplemente se lo utiliza de la misma forma en la que se lo haria desde cualquier otro lenguaje .NET teniendo en cuenta que los tipos importados se encuentran dentro del espacio de nombres “DotNET”.
El escribir nuestro código dentro de un bloque Namespace tiene por finalidad el poder mantener una especie de jerarquía. Para que no entendamos mejor el concepto de Namespace podríamos comparar los espacios de nombres con los directorios de un disco. En cada directorio tendremos ficheros que de alguna forma están relacionados, de esta forma no mezclaremos los ficheros de música con los de imágenes ni con los proyectos de Meta D++, por poner algunos ejemplos.
Pues lo mismo ocurre con las jerarquías creadas con los Namespace, de forma que podamos tener de alguna forma separados unos tipos de datos (clases, etc.) de otros. Siguiendo con el

ejemplo de los directorios, es habitual que dentro de un directorio podamos tener otros directorios, de forma que tengamos ficheros que estando relacionados con el directorio principal no queremos que se mezclen con el resto de ficheros. Pues con los Namespace ocurre lo mismo, podemos declarar bloques Namespace dentro de otro bloque existente. Esto se consigue definiendo un bloque dentro de otro bloque.

5.8.2 Cláusula Using

La cláusula Using se trata de una adopción sintáctica para hacer referencia para simplificar el uso de un recurso de la biblioteca de clases.
Para permitir hacer referencia a miembros de un espacio de nombres con nombres simples desde otro espacio de nombres se proporciona la directiva “using”.
Las directivas using sólo se pueden declarar en el encabezado de una Unidad de Traducción antes de la declaración de cualquier espacio de nombres. La sintaxis para la declaración de directivas using es la siguiente:

```
Using_Directive:
     using Complete_Class_Name ;
```

La directiva using se declara con la palabra clave “`using`” seguida de un nombre calificado y un punto y coma. El nombre calificado en la declaración de la directiva using denota el espacio de nombres al que se desea acceder con nombres simples.
La directiva using importa los nombres declarados en el espacio de nombres, especificado en la directiva, al alcance de la Unidad de Traducción actual. La directiva using sólo vuelve alcanzables los nombres declarado en el cuerpo del espacio de nombres importado pero no los nombres declarados en espacios de nombres anidados, ellos deben ser referenciados con el nombre calificado pudiéndose omitir el espacio de nombre importado.
La directiva using vuelve alcanzables los nombres en la Unidad de Traducción actual, pero no crea un espacio de declaraciones como lo hace la declaración de un espacio de nombres. Un nombre declarado dentro de un espacio de nombres en una unidad de traducción especifica ocultara a un nombre importado mediante una directiva using si ambos nombres son idénticos, en dicho caso se podrá referir al nombre oculto mediante la calificación completa del nombre.
La directiva using puede importar un espacio de nombre declarado en el programa actual Meta D++, o en alguna de las referencias externas del programa Meta D++ actual. Si el espacio de nombres declarado en la directiva using no es encontrado en el programa actual o en los módulos referenciados se produce un error de compilación.

```
// Ejemplo de directivas using:
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
using DotNET::System;

namespace Test{
   class A{
      static void Main(string^[] args){
         Console::WriteLine("Esto es una prueba");
      }
   }
}

// Se puede poner un ejemplo del mismo programa sin el using

import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";

namespace Test{
```

```
   class A{
      static void Main(string^[] args){
         DotNET::System::Console::WriteLine("Esto es una prueba");
      }
   }
}
```

5.8.3 Cláusula import

La directiva import permite importar tipos dependientes de la plataforma de implementación en un programa Meta D++. Las directivas import son procesadas por los Módulos de Salida Zoe durante el proceso de compilación del código Zoe de todo programa LayerD.
La directiva import se declara con la palabra reservada "`import`" seguida de un listado de literales de expresión.
El listado de literales de expresiones serán pasadas por el compilador Zoe al módulo de salida utilizado, es decir son los parámetros de importación de tipos que recibe el Módulo de Salida Zoe y por ende la sintaxis para las expresiones puede variar de un módulo de salida a otro.

```
// Ejemplo de directivas import:
// utilización de las directivas import (para plataforma .NET)
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   class Test{
   public:
      static void Main(string^[] args){
         Console::WriteLine("Bien realizado");
         string^ e = Console::ReadLine();
         int n = Convert::ToInt32(e);
      }
   }
}

//utilización de las directivas import para plataforma Java
import "java.lang.*", "platform=Java", "ns=Java";
using Java::java::lang;

namespace T{
   class Test{
   public:
      static void Main(string^[] args){
         System::@out.println("Bien realizado");
      }
   }
}
```

5.9 Herencia

El mecanismo de herencia es uno de los pilares fundamentales en los que se basa la programación orientada a objetos.
Cuando declaramos una clase e indicamos que hereda de otra, hacemos que nuestra clase "herede" todas las declaraciones de la clase ancestro, con la posibilidad de añadir nuevos miembros y modificar algunos heredados. Se dice en estos casos que heredamos tanto la estructura como el comportamiento. Para evitar la herencia múltiple, que no es muy aconsejable, el lenguaje Meta D++ permite la utilización de Interfaces. Estos tipos son muy similares a las clases, pero sólo permiten declaraciones de métodos, eventos y propiedades, pero para ninguno de estos miembros se permite la implementación en la clase que contiene la declaración

El lenguaje Meta D++ soporta tanto el modelo de herencia simple como el de herencia múltiple.

5.9.1 Herencia Simple

En el modelo de herencia simple cada clase puede heredar a lo sumo de otra clase, pero no de más de una clase.
El modelo de herencia simple es el preferido siempre que las necesidades de diseño del software se ajusten a él. El modelo de herencia simple es soportado por todas las clases de Módulos de Salida Zoe, por ello un programa que se ajuste a dicho modelo podrá compilarse sin limitaciones o cambios en cualquier plataforma soportada por la tecnología LayerD.
Si bien el lenguaje Meta D++ no proporciona una forma de explicitar el modelo de herencia a utilizar, es posible que una implementación provea mecanismos en su interfaz de compilación para exigir la aplicación y uso de sólo herencia simple, ello se puede implementar, por ejemplo, como argumentos de línea de comandos o mediante un archivo de configuración para el proyecto.

```
// Ejemplo :
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   class Alumno{
      string^ nombre;
      string^ apellido;
   public:
      Alumno(string^ nombre, string^ apellido){
         this.nombre = nombre;
         this.apellido = apellido;
      }
   }
       //utiliza la palabra inherits para heredar de la clase Alumno
   class Alumnorecibido inherits Alumno{
      bool graduado;
   public:
      Alumnorecibido(string^ nombre, string^ apellido): base(nombree, apellido){
         graduado = true;
      }
   }
   class Test{
   public:
      static void Main(string^[] args){
         Alumno^ a = new Alumno("Perez","Mariano");
         Console::WriteLine("se creo el alumno Mariano Perez");
         Console::ReadLine();
         Alumno^ b = new Alumno("Tenaglia","Nicolas");
      }
   }
}
```

5.9.2 Herencia Múltiple

El escribir un programa Meta D++ no es necesario especificar, y tampoco posible, que modelo de herencia, simple o múltiple, se utilizará, el modelo de herencia es determinado automáticamente por el compilador Zoe al analizar el programa y sus dependencias. Es preciso hacer notar, que aún cuando todas las clases en un desarrollo propio utilicen herencia simple, ellas pueden depender de clases que utilicen el modelo de herencia múltiple, lo cual resultara en la necesidad de aplicar forzosamente los mecanismos de herencia múltiple al construir el programa.

El modelo de herencia múltiple puede limitar las posibilidades de implementación de un programa LayerD, ya que no se exige que los Módulos de Salida Zoe más simples la implementen. Por dicha razón se recomienda, siempre que sea posible, evitar la utilización de herencia múltiple.
Cuando una clase tiene como clase base directa a más de una clase, se dice que dicha clase utiliza herencia múltiple y en tal caso un programa cliente de dicha clase también utilizará éste modelo de herencia múltiple.
En el modelo de herencia múltiple de Meta D++ por defecto todas las clases bases son virtuales. Si es preciso utilizar herencia de clases bases no virtuales se debe utilizar el modificador "nonvirtual" en las declaraciones de herencia de las clases para evitar la herencia de clases bases virtuales.

5.10 Interfaces

5.10.1 Concepto de Interfaz

Una interfaz es la definición de un conjunto de métodos para los que no se da la implementación, sino que se las define de manera similar a como se definen los métodos abstractos. Una interfaz puede verse como una forma especial de definir clases abstractas que tan sólo contengan miembros abstractos
Las interfaces son tipos por referencia, no puede crearse objetos de ellas sino sólo son de tipos que derivan de ellas y participan del polimorfismo.
Las interfaces sólo pueden tener como miembros métodos normales, eventos, propiedades e indicadores, pero no pueden incluir definiciones de campos, operadores, constructores, destructores o miembros estáticos. Todos los miembros de la interfaz son públicos y no se les pueden dar ningún modificador de acceso (ni siquiera public, lo supone).

5.10.2 Definición de Interfaz

La sintaxis general para definir una interfaz es la siguiente:

```
Interface <nombreinterface>  inherits  <interfacebase>
```

```
// Ejemplo:
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   public interface Ventana{
      void Incrementar();
   }
   class Estructura implements Ventana{
   public:
      int valor;
      void Incrementar(){
         valor = valor + 1;
      }
   }
   class Test{
   public:
      static void Main(string^[] args){
         Ventana^ o = new Estructura();
         o.Incrementar();
         Console::WriteLine(o.valor);
      }
   }
```

```
}
```

Una interfaz se declara con una serie de modificadores opcionales seguidos de la palabra clave "`interface`" y un identificador, el cual será el nombre de la interfaz, luego un listado opcional de interfaces bases con la palabra clave "`inherits`" antepuesta y finalmente el cuerpo de la interfaz donde se declaran los miembros de la interfaz encerrados entre llaves.

Una declaración de una interfaz introduce un nuevo nombre de tipo el cual se denomina *tipo de interfaz*, el nombre declarado para el tipo es el nombre de la interfaz. Una declaración de una interfaz crea un nuevo espacio de declaraciones que abarca el cuerpo de la interfaz.
La declaración de una interfaz no puede repetir modificadores de declaración y se prohíbe la declaración de más de un modificador de nivel de acceso.
El nivel de acceso de todos los miembros de una interfaz es público y no puede cambiarse.
Una interfaz heredar miembros de una interfaz ya existente de la misma forma en que una clase puede heredar miembros de otra clase. Para especificar las interfaces a partir de las cuales deseamos heredar miembros al declarar una interfaz debemos proporcionar un listado de una o más interfaces en la declaración de la interfaz utilizando la palabra clave "`inhertis`" y un listado de interfaces a heredar siguiendo la sintaxis siguiente:

```
Interface_Base_List:
     Interface_Base
     Interface_Base_List , Interface_Base

Interface_Base:
     Complete_Class_Name
```

Las interfaces nombradas en la declaración de herencia de una interfaz se denominan interfaces bases directas de la interfaz. Las interfaces bases directas de una interfaz deben ser al menos tan accesibles como la interfaz misma.
Una interfaz no puede heredar directa o indirectamente de si misma, ello producirá un error de compilación.
Las interfaces bases de una interfaz son sus interfaces bases directas y sus interfaces bases. La definición anterior implica que la herencia en interfaces, al igual que en las clases, es una propiedad transitiva. Una interfaz hereda todos los miembros de sus interfaces bases.

5.10.3 Implementación de Interfaz

Los miembros de interfaz son los miembros declarados en el cuerpo de implementación de la interfaz. Adicionalmente también son miembro de interfaz los miembros heredados de las interfaces bases. La sintaxis para la declaración de los miembros de interfaz es la siguiente:

```
Interface_Decl_Block:
     Interface_Member_Decl
     Interface_Decl_Block Interface_Member_Decl

Interface_Member_Decl:
     Function_Decl
     Property_Decl
     Indexer_Decl
     ClassFactory_Call
     FPointer_Decl
```

Los miembros de una interfaz pueden ser métodos, propiedades e indexadores. Una interfaz debe declarar cero o más miembros. Todos los miembros de una interfaz son siempre públicos, por dicha razón no se permite utilizar especificadores de nivel de acceso en el cuerpo de una interfaz a diferencia de las clases y estructuras.
Una interfaz no puede declarar campos miembros, constructores, destructores o tipos anidados, la única excepción es la posibilidad de declarar un tipo de puntero a función.
Cada método y propiedad miembro de interfaz introduce un nombre dentro del espacio de declaraciones de la interfaz. Los nombres introducidos mediante los miembros de interfaz deben cumplir con los siguientes requisitos:
El nombre de los métodos declarados en una interfaz debe diferir del nombre de todas las propiedades declaradas en la interfaz y su nombre, tipo, orden y cantidad de parámetros (firma) debe diferir de cualquier otro método declarado en la interfaz.
El nombre de una propiedad debe diferir del nombre de cualquier otra interfaz declarada en la interfaz.
Los miembros de una interfaz pueden ocultar miembros heredados de una interfaz base sin producir un error de compilación. En dicho caso se requiere utilizar la palabra reservada "`new`" en la declaración del miembro de la interfaz. Es importante tener en cuenta que el operador "`new`" puede ser ignorado al construir el programa en algunos Módulos de Salida Zoe y forzar al sobre-escritura de métodos.
Adicionalmente se deben tener en cuenta las siguientes indicaciones:
No se puede proporcionar un cuerpo de implementación, en su lugar se requiere sólo un punto y coma.
No se permite utilizar los modificadores de declaración siguientes: "`virtual`", `"nonvirtual", "extern", "abstract", "static", "override".`
Las propiedades en las interfaces se declaran siguiendo la sintaxis especificada en (), para los métodos, con las siguientes limitaciones:
No se puede proporcionar un cuerpo de implementación para los accesos de la propiedad en su lugar se requiere sólo un punto y coma, como ser "`get;"` o `"set;"`.
No se permite utilizar los modificadores de declaración siguientes: `"virtual", "nonvirtual", "extern", "abstract", "static", "override".`

6 CONROL DE EXCEPCIONES (MANEJO DE ERRORES)

Las excepciones son el mecanismo para tratar los errores que se generan en tiempo de ejecución, por ejemplo se puede producir divisiones por cero, lectura de archivos no disponibles, etc.
Una excepción es un objeto que cuenta con campos que describen las causas del error y a cuyo tipo suele dársele un nombre que resuma claramente su causa. Cuando se usan excepciones siempre se asegura que el programador trate toda excepción que pueda producirse o que, si no lo hace, se aborde la ejecución de la aplicación mostrándose un mensaje indicando dónde se ha producido el error.
Existen distintos tipos de excepciones que manejan los errores más comunes que pueden producirse durante la ejecución de una aplicación. A continuación se mencionan las mismas:

6.1 Lanzamiento de Excepciones

Para lanzar las excepciones es necesario utilizar la cláusula throw.
La sintaxis es la siguiente:

```
   throw <objetoExcepciónALanzar>;
```

Por ejemplo, para lanzar una excepción de tipo `DivideByZeroException` se podría hacer:

```
   throw new DivideByZeroException();
```

6.2 Captura de Excepciones

Una vez lanzada una excepción es posible escribir código que se encargue de tratarla. Por defecto, si este código no se escribe la excepción provoca que la aplicación aborte mostrando un mensaje de error en el que se describe la excepción producida (información de su propiedad `Message`) y dónde se ha producido. Así, dado el siguiente código fuente de ejemplo:

```
// Ejemplo:
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T
{
   class PruebaExcepciones
   {
      static void Main()
      {
         int c=0;
         double division;
         int d=2;
         division= d/c;
         Console::WriteLine("el resultado de la divisíon es:" + division);
      }
   }
}
```

Al Ejecutar el código anterior dará error ya que intentará dividir por cero. Para evitar que se produzca un error es necesario tratar la excepción de la siguiente forma:
Si se desea tratar la excepción hay que encerrar la división dentro de una instrucción try con la siguiente sintaxis:

```
   try
      <instrucciones>
   catch (<excepción1>)
      <tratamiento1>
   catch (<excepción2>)
      <tratamiento2>
   ...
   finally
      <instruccionesFinally>
```

El significado de **try** es el siguiente: si durante la ejecución de las <instrucciones> se lanza una excepción de tipo <excepción1> (o alguna subclase suya) se ejecutan las instrucciones <tratamiento1>, si fuese de tipo <excepción2> se ejecutaría <tratamiento2>, y así hasta que se encuentre una cláusula **catch** que pueda tratar la excepción producida. Si no se encontrase ninguna y la instrucción **try** estuviese anidada dentro de otra, se miraría en los **catch** de su **try** padre y se repetiría el proceso. Si al final se recorren todos los **try** padres y no se encuentra ningún **catch** compatible, entonces se buscaría en el código desde el que se llamó al método que produjo la excepción. Si así se termina llegando al método que inició el hilo donde se produjo la excepción y tampoco allí se encuentra un tratamiento apropiado se aborta dicho hilo; y si ese hilo es el principal (el que contiene el punto de entrada) se aborta el programa y se muestra el mensaje de error con información sobre la excepción lanzada ya visto.

El bloque finally es opcional, y si se incluye ha de hacerlo tras todas los bloques catch. Las <instruccionesFinally> de este bloque se ejecutarán tanto si se producen excepciones en <instrucciones> como si no. En el segundo caso sus instrucciones se ejecutarán tras las <instrucciones>, mientras que en el primero lo harán después de tratar la excepción pero antes de seguirse ejecutando por la instrucción siguiente al try que la trató. Si en un try no se encuentra un catch compatible, antes de pasar a buscar en su try padre o en su método llamante padre se ejecutarán las <instruccionesFinally>.

Sólo si dentro de un bloque finally se lanzase una excepción se aborta la ejecución del mismo. Dicha excepción sería propagada al try padre o al método llamante padre del try que contuviese el finally. Aunque los bloques catch y finally son opcionales, toda instrucción try ha de incluir al menos un bloque catch o un bloque finally.

```
// Ejemplo:
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
import "System", "platform=DotNET", "ns=DotNET", "assembly=System";
using DotNET::System;

namespace T{
   class PruebaExcepciones{
      static void Main(){
         try {
            double c = 0.0;
            double division;
            double d = 2.0;
            division = d / c;
         }
         catch (DivideByZeroException^ error){
            Console::WriteLine(division);
            Console::WriteLine("No se puede dividir por cero");
         }
      }
   }
}
```

Segunda Parte

Conceptos de Tiempo de Compilación y Guía paso a paso de programación en Tiempo de Compilación

7 TIEMPO DE COMPILACIÓN VS. TIEMPO DE EJECUCIÓN

LayerD provee dos tiempos de ejecución diferenciados. El tiempo de ejecución es conocido por todos y refiere simplemente al momento de la ejecución del programa. Por otro lado LayerD nos permite realizar procesamiento durante el proceso de compilación de un programa.
Para programar el procesamiento deseado durante el tiempo de compilación se utilizan clases especiales denominadas “Classfactorys”.

7.1 Teoría rápida sobre Classfactorys

Las Classfactorys son simplemente clases destinadas a programar el procesamiento de tiempo de compilación de un programa LayerD. Durante la compilación no pueden evaluarse todos los valores, por otro lado, es posible acceder a construcciones que ya no existen en tiempo de ejecución.
Durante la compilación de un programa LayerD se puede acceder al código del programa que se está compilando (reflexión en tiempo de compilación) ya sea para analizarlo o modificarlo (siempre que la classfactory posea los permisos necesarios). Debido a esto las classfactorys permiten utilizar tipos de datos y objetos especiales.
Los tipos de datos especiales utilizables por las classfactorys son los siguientes:

Tipo	Descripción	Ejemplo

Expresión	Las variables de tipo expresión representan una expresión en el código fuente.
	Los miembros de classfactorys pueden tomar como argumento expresiones o sea de tipo expresión (como en el caso de campos o propiedades).
	Para declarar una variable como de tipo expresión debe utilizarse el modificador de declaración “exp” como en “exp int campo;” este modificador indica al compilador que lo que queremos guardar en dicha variable es una expresión y no su valor.
	Un tipo de dato expresión tiene asociado un tipo de expresión, por ejemplo “exp int varInt” tiene asociado el tipo int, por tanto sólo podrá almacenar expresiones de tipo int o de un tipo convertible a int.
	Las variables de tipo expresión son equivalentes al tipo “LayerD::CodeDOM::XplExpression^”.
		`factory class F{`
		
		`exp int fieldExp;`
		
		`exp void func(exp int arg1){`
		`}`
		
		`}`

Tipo	Las variables de tipo “type” (o sea tipo) representan un tipo de datos. Es posible tomar como argumento tipos o retornarlos desde funciones especialmente de “constructores de tipo”.
	El tipo intrínseco “type” es equivalente al tipo “LayerD::CodeDOM::XplType^”.
		`type varMyType;`
		
		`type Func(type arg1, type arg2){`
		`}`

Bloques	Las variables de tipo “block” nos permiten tomar como argumento de funciones bloques de código fuente para ser procesado. Cuando es utilizado como argumento de función, el tipo block debe ser el último argumento de la función y no puede poseerse más de un argumento de tipo block.
	El tipo intrínseco “block” es equivalente al tipo “LayerD::CodeDOM::XplFunctionBody^”.
		`block field;`
		`exp void func(block arg){`
		`field = arg;`
		`}`

Identificadores	Si todo lo que necesitamos tomar como argumento de una función es un identificador y no una expresión podemos utilizar el modificador de tipo “iname”, si declaramos un argumento “iname void argName” podremos tomar como argumento un identificador de cualquier tipo, incluyendo un identificador no definido, si declaramos el argumento “iname int argName” la función tomara un argumento que sea un identificador, pero el identificador deberá estar declarado y ser de tipo entero en este caso.
	El modificador de tipo intrínseco “iname” es equivalente al tipo “LayerD::CodeDOM::XplIName^”.
	Es posible utilizar la clase XplIName para crear identificadores que necesitemos utilizar en expresiones “writecode”, por ejemplo “XplIName id = new XplIName()” crea un iname de tipo “void” (sin tipo) con un nombre de identificador único, lo cual es útil para generar nombres de identificadores.
		`void func(iname void arg,`
		`iname int arg2){`
		
		`}`

Los objetos especiales a los cuales tienen acceso las classfactorys son los siguientes:

Objeto	**Descripción**	**Ejemplo**

context	El objeto “context” representa el contexto de llamada actual a un miembro de una classfactory.
	El objeto “context” posee miembros útiles como las propiedades “CurrentNamespace”, “CurrentClass”, etc., que representan las construcciones más inmediatas que engloban al contexto de llamada del miembro.
	El objeto “context” se utiliza para analizar el contexto de llamada o para utilizarlo como referencia al insertar un nodo.
		`void func (){`
		`…`
		`//representa la clase en la cual es llamada a la función func`
		`context.CurrentClass.`
		`Children().`
		`InsertAtEnd(member);`
		`…`
		`}`

currentDTE	El objeto “currentDTE” representa el código fuente que se está compilando actualmente y permite realizar análisis y modificación (siempre que se posean los permisos adecuados). El objeto “currentDTE” es de tipo “LayerD::CodeDOM::XplDocument^”, por tanto pueden utilizarse todos los miembros de esta clase.	`//Busca todas las funciones en el código currentDTE. FindNodes(“/@XplFunction”);`
compiler	El objeto “compiler” representa la instancia actual del compilador zoe que esta compilando el modulo. Es de tipo “LayerD::ZoeCompiler::ZoeCompilerCore^”. Se utiliza para agregar errores, advertencias o consultarlos, entre otras cosas.	`//Agrega un error al proceso de compilación compiler.Errors.Add(“error”, errorNode);`

7.2 Generar e Inyectar código en tiempo de compilación

Para generar código en tiempo de compilación se deben seguir dos pasos sencillos:

- Generar el código que se quiere inyectar en la memoria
- Insertar el código en memoria en el lugar deseado dentro del código del programa que se está compilando.

Para generar el código en memoria se lo puede realizar de tres formas:

- Usando las clases dentro del espacio de nombres LayerD::CodeDOM (o DotNET::LayerD::CodeDOM de acuerdo a la implementación del compilador Zoe utilizado).
- Usando la expresión “writecode”.
- Usando una combinación de las anteriores.

Normalmente se encontrará útil combinar las dos primeras formas de generar código, sobretodo utilizaremos la expresión “writecode” para generar el grueso del código y las clases del CodeDOM para complementar.

7.2.1 El CodeDOM

En LayerD se denomina “CodeDOM” al conjunto de clases declaradas dentro del espacio de nombres “LayerD::CodeDOM” (según la implementación del compilador Zoe puede estar contenido dentro de otro espacio de nombres como “DotNET::LayerD::CodeDOM”, “Java::LayerD::CodeDOM”, etc.).
Las clases dentro del CodeDOM representan porciones de código fuente Zoe en memoria, por ejemplo clases, funciones, expresiones, tipos, etc. Todas las clases que representan alguna porción de código fuente derivan de XplNode la cual es la clase base para todos los nodos que representan un fuente Zoe.
En memoria un fuente Zoe es representado con una estructura de árbol y las clases dentro del CodeDOM representan las ramas y hojas dentro del árbol.
Como clases más importantes dentro del CodeDOM pueden nombrarse las siguientes:

Clase	**Descripción**

`XplNode`	Es la clase base de todas las clases que forman parte de un documento Zoe en memoria.
`XplDocument`	Representa el nodo inicial de un Documento Zoe en memoria. Contiene un "DocumentData" que contiene meta-información e información de configuración y un "DocumentBody" donde se encuentra el código funcional del fuente.
`XplNamespace`	Representa un espacio de nombres Zoe, para establecer u obtener el nombre se utiliza "set_name" o "get_name".
`XplClass`	Representa una clase, interface, enumeración u estructura Zoe. Por defecto representa una clase.
`XplFunction`	Representa una función miembro de una clase Zoe.
`XplProperty`	Representa una propiedad miembro de una clase Zoe.
`XplField`	Representa un campo miembro Zoe.
`XplType`	Representa un tipo de datos, es una estructura recursiva. Se utiliza para representar el tipo de datos "type".
`XplFunctionBody`	Representa un cuerpo de función, contiene instrucciones. Se utiliza para representar el tipo de datos "block".
`XplIName`	Representa un "iname" o identificador a los fines de ser utilizado por el compilador Zoe, pero no representa en si una porción de código fuente.
`XplClassMembersList`	No se utiliza como parte de un programa Zoe, sino para agrupar miembros de clase en memoria.
`XplExpression`	Representa un contenedor de algún tipo de expresión. Por ejemplo, puede contener una expresión literal, unaria, binaria, etc. Es el tipo de datos utilizado para representar los tipos de datos expresión "exp".
`XplNodeList`	Es una lista de nodos de tipo que se utiliza para almacenar cualquier tipo de nodo derivado de XplNode.

Como XplNode es la clase base de todos los nodos en el CodeDOM su funcionalidad se comparte con la mayor parte de los tipos dentro del CodeDOM. Puede pensar a una instancia de XplNode como la representación de un nodo Xml del fuente Zoe, luego cada clase derivada representa una porción de código fuente Zoe diferente en memoria. La funcionalidad principal de XplNode se detalla en la siguiente tabla:

Miembro	Descripción
`get_ElementName () y set_ElementName(string^)`	Obtiene o estable el nombre del elemento XML.
`Children()`	Devuelve un XplNodeList con los nodos hijos. Sólo para elementos que puedan contener una lista variable de nodos hijos como clases, bloques, etc. Si el elemento no puede contener una lista variable de hijos, como una expresión, devuelve nulo.
`get_Parent()`	Devuelve el padre del nodo, nulo si no tiene padre.
`get_Content()`	Devuelve el contenido del nodo para nodos de tipo “contenedor” como nodos de tipo XplExpression devuelve el nodo de expresión específico. Nulo si no contiene ningún elemento.
`set_Value(string^)`	Establece el contenido del nodo al string pasado como argumento (sólo sirve para nodos simples de tipo XplNode).
`get_StringValue()`	Obtiene el valor string almacenado en el nodo. Sólo se sirve con nodos de tipo XplNode que contengan un valor de tipo string (los nodos simples pueden contener otros tipos de valores como enteros, flotantes o fechas).
`FindNodes(“/n”)`	Devuelve una lista de nodos (un XplNodeList) que coincidan con la cadena de búsqueda pasada como argumento. Vea más abajo para el formato de la cadena de búsqueda.
`FindNode(“/n(id)”)`	Devuelve el primer nodo encontrado que cumpla con el patrón pasado como argumento o nulo si no encuentra un nodo.
`CurrentNamespace`	Devuelve el espacio de nombre más inmediato que contiene al nodo actual o nulo si el nodo actual no se encuentra dentro de un espacio de nombres.
`CurrentClass`	Devuelve la clase más inmediata que contiene al nodo actual o nulo si no se encuentra dentro de una clase.
`CurrentFunction`	Devuelve la función más inmediata que contiene al nodo actual o nulo si no se encuentra dentro de una función.
`CurrentProperty`	Devuelve la propiedad más inmediata que contiene al nodo actual o nulo si no se encuentra dentro de una propiedad.
`CurrentBlock`	Devuelve el bloque más inmediato que contiene al nodo actual o nulo si no se

	encuentra dentro de un bloque.
`CurrentExpression`	Devuelve la expresión más inmediata que contiene al nodo actual o nulo si no se encuentra dentro de una expresión.
`ChildNodes()`	Devuelve una colección con todos los nodos hijos. Si el elemento no posee nodos hijos devuelve una colección vacia.
`string^[] Attributes()`	Devuelve una matriz con todos los atributos XML del tipo del elemento.
`string^ AttributeValue(string^)`	Devuelve el valor de el atributo indicado en el parámetro convertido a tipo string como es representado en XML cuando se almacena un documento Zoe.
`set_doc(string^) y get_doc()`	Establece y retorna un comentario simple asociado al elemento.
`string^ get_ldsrc()`	Devuelve un string indicando las líneas/columnas de inicio y/o fin en el fuente original. Puede devolver una cadena vacía en cuyo caso indica que el nodo no posee información de fuente de origen, en dicho caso deberá examinarse los nodos padres más cercanos.
`CodeDOMTypes get_TypeName()`	Devuelve un valor de enumeración con el nombre del tipo del nodo, por ejemplo “XplNode”, “XplClass”, etc. Puede utilizarlo opcionalmente a la identificación de tipos soportada por el lenguaje de alto nivel.
`XplNode^ Clone()`	Obtiene una copia profunda de un nodo.
`Write(XplWriter^)`	Guarda el nodo y todos los hijos usando la instancia de XplWriter recibida como argumento.
`XplNode^ Read(XplReader^)`	Carga en memoria el nodo a partir de la instancia de XplReader recibida como argumento.

La clase XplNodeList es utilizada por las clases del CodeDOM donde se requiere una lista de nodos de cualquier longitud y donde en general dicha lista puede poseer varios tipos de nodos del CodeDOM.
Los miembros más importantes de XplNodeList son:

Miembro	**Descripción**
`InsertAtEnd(XplNode^)`	Inserta un nodo al final de la lista. También existe una sobrecarga que acepta una XplNodeList como argumento, en cuyo caso inserta todos los nodos contenidos por la lista recibida.
`InsertAtBegin(XplNode^)`	Inserta un nodo al principio de la lista. Existe

	otra sobrecarga que acepta un XplNodeList.
`InsertBefore(XplNode^ reference, XplNode^ newNode)`	Inserta newNode antes del nodo reference. Si el nodo reference no existe no inserta el newNode. Existe otra sobrecarga que acepta un XplNodeList.
`InsertAfter(XplNode^ reference, XplNode^ newNode)`	Inserta newNode después del nodo reference. Si el nodo reference no existe no inserta el newNode. Existe otra sobrecarga que acepta un XplNodeList.
`FirstNode()`	Retorna el primer nodo de la lista e inicializa el iterador interno de la lista.
`NextNode()`	Obtiene el nodo siguiente en la iteración actual, si no existen más nodos (esta al final de la lista) retorna nulo.
`GetLength()`	Retorna la cantidad de nodos en la lista.
`XplNodeList::CopyNodesAtEnd(XplNodeList^ source, XplNodeList^ target)`	Copia los nodos dentro de la lista source al final de la lista target.

Utilizando instancias de las clases del CodeDOM se puede crear en memoria porciones de código fuente e incluso programas completos. Es preciso aclarar que todo código Zoe representado en memoria como un documento es implementado con las clases del CodeDOM, por ello es muy útil para todo programador de classfactorys conocer el CodeDOM.

7.2.2 La expresión “writecode”

En lugar de crear código fuente en memoria utilizando las clases del CodeDOM se puede utilizar la expresión “writecode”. La expresión writecode toma como argumento una expresión o un bloque de código dependiendo de la forma utilizada, y genera en memoria el código pasado como argumento utilizando las clases del CodeDOM, es decir la expresión writecode realiza automáticamente el trabajo que deberíamos de hacer a “mano” utilizando las clases del CodeDOM. Adicionalmente la expresión writecode realiza reemplazos en su bloque argumento reemplazando identificadores marcados con “$” por el contenido de la variable nombrada.
La expresión writecode posee tres formas sintácticas diferentes:

Ejemplo	**Descripción**
`//Devuelve un objeto XplExpression` `writecode( $argumentExp + 5 * var )`	Forma de Expresión En la forma de expresión la instrucción writecode siempre devuelve un objeto de tipo “XplExpression^”.

`//Devuelve un objeto XplClass`
`writecode{`
`  class $className{`
`  void func () { }`
`  }`
`}`
 	
Ó
 	
`//Devuelve un objeto XplDocumentBody`
`writecode{`
`  using Zoe;`
`  namespace test{`
	Forma de Bloque simple
	En la forma de bloque simple la expresión writecode devuelve un objeto del tipo “XplClass^”, “XplFunctionBody^” o “XplDocumentBody^” dependiendo del contenido del bloque argumento.

}
}
 	
Ó
 	
//Devuelve un objeto XplFunctionBody
writecode{
int localvar = 0;
for(int n=0; n<$arg.Count; n++){
localvar++;
}
}	

//Devuelve un objeto XplClassMembersList
//Notar que se requiere "{%" y "%}" para
//el caso de generar miembros de clase
writecode{%
int field1;
float field2;
void func($t1 arg1, int arg2) { }
%}
	Forma de Bloque de miembros de Clase
	En esta forma sintáctica la expresión writecode devuelve un objeto del tipo "XplClassMembersList^" el cual contiene una lista de miembros de clase.

Para indicarle a la expresión writecode que deseamos reemplazar un identificador por el contenido de una variable debemos marcar el identificador con el signo de dólar al comienzo, por ejemplo "$miVariable". A continuación se muestra un ejemplo:

```
members = writecode
{%
private:
        $fieldType $internalFieldName;
public:
        $fieldType property $fieldName{
                get{
                  return $internalFieldName;
                }
                set{
                  $internalFieldName = value;
                }
        }
%};
```

En el ejemplo superior "fieldType", "internalFieldName" y "fieldName" son variables en el alcance de la expresión writecode. Los tipos que se pueden reemplazar en una expresión writecode son: type o XplType^, block o XplFunctionBody^, XplExpression^ o tipos "exp", XplIName^ o tipos "iname", constantes.

7.2.3 Retornar expresiones y tipos desde una función miembro de classfactory

Para retornar expresiones desde un miembro de una classfactory se puede utilizar la instrucción "return" ordinaria como en cualquier otra función.
Ejemplos:

```
factory class Example{
public:
       static exp string^ Add(exp int arg1, exp int arg2){
              return writecode( "Result: " + ($arg1+$arg2).ToString() );
       }
}
```

Para utilizar la función superior en un programa cliente lo hacemos como si se tratara de cualquier otra función:

```
Console::WriteLine( Example::Add(5, 1024) );
```

También es posible retornar expresiones utilizando variables del tipo XplExpression^. Por ejemplo:

```
factory class Example{
public:
	exp string^ Add(exp int arg1, exp int arg2){
		XplExpression^ resExp = null;
		resExp = writecode( "Result: " + ($arg1+$arg2).ToString() );
		return resExp;
	}
}
```

El ejemplo superior muestra como los tipos declarados con el modificador "exp" son equivalentes a una referencia al tipo XplExpression.
Para retornar tipos a código cliente podemos utilizar los "Constructores de Tipos". Los constructores de tipos se escriben igual que los constructores ordinarios pero se especifica que retornan un tipo, como se muestra a continuación:

```
factory class MyString {
public:
	type MyString(){
		return gettype(string^)
	}
}
```

Al utilizar el tipo MyString en un programa cliente el resultado será que cada ocurrencia de una declaración del tipo "MyString" se reemplazará por el tipo "string^", ejemplo:

```
MyString myVar = "Hola Mundo";
MyString[] myArray = new MyString[] = { "Zoe", "Meta D++" };
```

El ejemplo superior es equivalente a escribir las siguientes líneas:

```
string^ myVar = "Hola Mundo";
string^[] myArray = new string^[] = { "Zoe", "Meta D++" };
```

Los constructores de tipos se utilizan para desarrollar classfactorys que encapsulen la diferencia de tipos entre diversas plataformas o entornos de ejecución. También existe una variante de los constructores de tipos que permite tomar argumentos, para más información consulte los manuales de Meta D++, Zoe o visite el sitio web http://layerd.net.
El tipo "type" es equivalente al tipo "XpType^", como se muestra a continuación:

```
factory class MyType{
	static XplType^ refType;
public:
	static exp void SetType(type argType){
		refType = argType;
		return null;
	}
	type MyType(){
		if(refType==null)
			return gettype(int);
		else
			return refType;
	}
}
```

El ejemplo superior retorna entero si no se estableció un tipo llamando a SetType, de lo contrario retorna el tipo establecido en SetType.

7.2.4 El objeto “context”

El objeto “context” representa el contexto de llamada actual en un miembro de una classfactory. El objeto context es de tipo XplNode por tanto pueden accederse a todos los miembros de dicha clase.
Ejemplo:
La classfactory:

```
namespace Zoe::Examples{

factory class ContextSensitive{
public:
	exp void WhereAmI(){
		if(context.CurrentNamespace!=null && context.CurrentClass==null)
			Console::WriteLine( “You are on a namespace body.” );
		if(context.CurrentClass!=null && context.CurrentFunction==null &&
context.CurrentProperty==null)
			Console::WriteLine( “You are on a class body.” );
		if(context.CurrentBlock!=null)
			Console::WriteLine( “You are on a block.” );
		return null;
	}
}

}
```

El cliente:

```
using Zoe::Examples;

namespace N{
	ContextSensitive::WhereAmI();
	class A{
		ContextSensitive::WhereAmI();
		void Func(){
			ContextSensitive::WhereAmI();
		}
	}
}
```

7.2.5 El objeto “currentDTE”

El objeto “currentDTE” representa el fuente actual y refiere al “Documento de Tiempo de Ejecución” del tiempo de compilación en curso. El contenido del objeto currentDTE es la suma de todos los fuentes que componen el programa que se está compilando actualmente, por tanto al realizarse análisis o búsquedas en dicho objeto estamos realizando dichas acciones en el código de todo el programa, sin embargo esto no incluye los documentos importados por los generadores de código Zoe (por ejemplo los tipos importados usando la directiva import).

7.2.6 El objeto “compiler”

El objeto “compiler” representa la instancia actual del compilador Zoe. Con el objeto compiler es posible consultar los errores y advertencias del proceso de compilación actual, también es posible agregar nuevos errores o advertencias y consultar la información de tipos recopilada durante el análisis semántico. Ejemplo:

```
// add persistent compilation error
Error^ newError = new Error("Error processing xml file. " + error.Message,context);
newError.set_PersistentError(true);
compiler.Errors.AddError(newError);
```

8 GUÍA PASO A PASO DE PROGRAMACIÓN EN TIEMPO DE COMPILACIÓN (CON CLASSFACTORYS)

8.1 Classfactorys de Menor a Mayor

La siguiente guía mostrara una serie de ejemplos partiendo desde lo más sencillo a temas más complejos. En todos los casos se tendrán dos archivos. Uno para la classfactory y otro para el cliente de manera de mostrar el consumo y prueba.
Para probar los ejemplos primero se deberá compilar la classfactory y agregarla a la librería de classfactorys usando el modificador "-ae" del compilador Zoe, y luego compilar el cliente.

8.1.1 Hola Mundo con una Classfactory

El código de la classfactory es la siguiente:

```
using Zoe;
using DotNET::LayerD::ZoeCompiler;
using DotNET::LayerD::CodeDOM;

namespace Test{
	public factory class HelloWorld{
	public:
		static exp string^ SayHello(){
			return writecode( "Hello World" );
		}
	}
}
```

El código del cliente:

```
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
using DotNET::System;

namespace Test {
	public class App{
	public:
		static void Main(string^[] args){
			Console::WriteLine( HelloWorld::SayHello() );
		}
	}
}
```

En lo subsiguiente se omitirán las instrucciones Using, Import y la declaración del espacio de nombres, pero deberá incluirse en todos los ejemplos para poder compilarlos, aquí se los omitirá por comodidad.
Para compilar el ejemplo anterior guarde el código de la classfactory en un archivo (aquí "TestTemplates.dpp") y el cliente en otro archivo ("Test.dpp") luego compile con los siguientes comandos:

```
c:\layerd\bin>metadppc.exe TestTemplates.dpp
c:\layerd\bin>zoec.exe TestTemplates.zoe -ae

c:\layerd\bin>metadppc.exe Test.dpp
c:\layerd\bin>zoec.exe Test.zoe
```

De la classfactory destacamos:

```
		static exp string^ SayHello(){
			return writecode( "Hello World" );
		}
```

Las palabras reservada “exp” antes de la declaración de tipos indicamos que en realidad queremos especificar una expresión de determinado tipo, en este caso indicamos que queremos devolver una expresión de tipo “string^”, es decir una referencia a string o lo que es lo mismo un equivalente al tipo de datos string en C# o Java – lenguajes en los cuales siempre son por referencia las cadenas – luego la palabra reservada “writecode” crea en memoria una porción de código – en este caso una expresión – que finalmente es retornada por la instrucción return.
El resultado es que al ejecutarse la llamada a SayHello durante el tiempo de compilación del programa cliente se retornara la expresión `“Hello World”` y se insertará en el lugar de la llamada a función.

8.1.2 Classfactorys y Compilación Interactiva de programas LayerD

Para aprender a programar Classfactorys y utilizar las características de tiempo de compilación de LayerD es útil conocer una versión especial de classfactorys denominadas “interactivas”.
Las llamadas classfactorys interactivas son ejecutadas cuando compilamos un programa cliente en “modo interactivo”, el modo de compilación interactivo de un programa LayerD funciona de la siguiente forma:

- El compilador Zoe compila el programa normalmente y ejecuta en tiempo de compilación todas las llamadas a Classfactorys Interactivas.
- Al resultado del primer tiempo de compilación lo convierte al código fuente LayerD de alto nivel sobre-escribiendo el fuente original.

La utilidad y aplicación de las classfactorys interactivas es muy variada. Ya que permite programar comportamiento muy complejo que normalmente se encuentran en los IDEs de los lenguajes como refactoring, análisis de código, utilización de snippets de código, ejemplos, asistentes, diseñadores visuales, etc. Pero lo más importante es que nos permite programar todas estas cosas de manera independiente al IDE, en nuestro propio lenguaje de alto nivel y si queremos independiente de la plataforma utilizando las características de LayerD. También cabe recordar que las classfactorys interactivas pueden ser consumidas por cualquier lenguaje LayerD cliente que soporte generación inversa (de código Zoe al código del meta lenguaje) como por ejemplo Meta D++.
Una classfactory interactiva es prácticamente idéntica a una classfactory ordinaria, simplemente debe cambiarse en la declaración el modificador “factory” por el modificador “interactive”, ej:

```
using Zoe;
using DotNET::LayerD::ZoeCompiler;
using DotNET::LayerD::CodeDOM;

namespace Test{
	public interactive class HelloWorld{
	public:
		static exp string^ SayHello(){
			return writecode( “Hello World” );
		}
	}
}
```

Luego se compila la extensión como cualquier otra classfactory, por ejemplo:

```
c:\layerd\bin>metadppc.exe TestTemplates.dpp
c:\layerd\bin>zoec.exe TestTemplates.zoe –ae
```

En los programas cliente se las utilizan de forma idéntica a una classfactory ordinaria:

```
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
using DotNET::System;

namespace Test {
	public class App{
	public:
		static void Main(string^[] args){
			Console::WriteLine( HelloWorld::SayHello() );
		}
	}
}
```

Luego sólo debemos ejecutar el compilador Zoe en "modo interactivo" – utilizando la opción `"-i"` del compilador – al compilar el programa cliente:

```
c:\layerd\bin>metadppc.exe Test.dpp
c:\layerd\bin>zoec.exe Test.zoe -i
```

Como resultado de la ejecución de la classfactory interactiva anterior nuestro programa cliente será modificado como sigue:

```
import "System", "platform=DotNET", "ns=DotNET", "assembly=mscorlib";
using DotNET::System;

namespace Test {
	public class App{
	public:
		static void Main(string^[] args){
			Console::WriteLine( "Hello World" );
		}
	}
}
```

Si asociamos el proceso de compilación interactiva a un macro de nuestro IDE preferido será muy sencillo llamarlo usando un atajo de teclado.
En los ejemplos siguientes resultara muy útil para el estudiante utilizar la compilación interactiva para ir verificando el funcionamiento de las classfactory. También es siempre útil utilizar este modo de compilación para depurar classfactorys complejas que estemos desarrollando (téngase en cuenta que es posible depurar paso por paso la ejecución de classfactorys si es necesario utilizando depuradores estándar para la plataforma en la cual se ejecute el compilador Zoe).
A modo de recordatorio es importante comprender que no es una buena práctica de programación abusar de classfactorys interactivas para inyectar código que después modificaremos "a mano", precisamente la función de las classfactorys (ordinarias, no las interactivas) es encapsular código y permitir al programador de alto nivel abstraerse de las complejidades de la implementación. Por tanto, sólo deben usarse classfactorys interactivas para desarrollar componentes tipo RAD útiles, como ejemplos de utilización de librerías, asistentes para clases, refactoring de código, diseñadores visuales y similares utilidades que normalmente estarían enlazadas y dependientes de un IDE en particular. Consulte otros documentos en http://layerd.net para más discusiones e información sobre classfactorys interactivas.

8.1.3 Hola Mundo con argumento constante

El código de la classfactory es la siguiente:

```
public factory class HelloWorld{
public:
	static exp string^ SayHello(string^ constantArgument){
		return writecode( "Hello World" + $constantArgument );
	}
}
```

El código del cliente:

```
public class App{
public:
	static void Main(string^[] args){
		Console::WriteLine( HelloWorld::SayHello( " from Argentina." ) );
	}
}
```

Compile el ejemplo de la misma manera y corra el programa cliente.

El cambio desde el ejemplo anterior es muy simple y se limita a agregar un argumento a la función `SayHello` de tipo `string^` y utilizarlo en la expresión `writecode` con el prefijo `"$"`. De esa forma es posible pasar un valor constante a la función de la classfactory cuando es llamada. En las expresiones `writecode` al utilizar identificadores precedidos por `"$"` estamos indicando que queremos reemplazar el identificador por el contenido de una variable, si dicha variable no existen en el alcance o no es de un tipo indicado se generará un error al intentar compilar la classfactory.

El resultado es que al ejecutarse la llamada a SayHello durante el tiempo de compilación del programa cliente se retornara la expresión `"Hello World from Argentina."` y se insertará en el lugar de la llamada a función.

Un concepto importante que se desprende del ejemplo se aprende al intentar en el programa cliente lo siguiente:

```
public class App{
public:
	static void Main(string^[] args){
	    Console::WriteLine( HelloWorld::SayHello( " from Argentina." + args[0] ) );
	}
}
```

Al intentar compilar el programa cliente superior se producirá un error indicando que no es posible evaluar la expresión `" from Argentina." + args[0]`, lo cual es correcto ya que debe recordarse que las llamadas a miembros de classfactorys se procesan en tiempo de compilación – esto es mientras se está compilando el programa – y por tanto no es posible evaluar el valor de expresiones que posean valores "de tiempo de ejecución" como en este caso la variable "args". Nótese que sencillamente no se posee un valor para args ya que el programa todavía no se ejecuto y por tanto no es posible determinar un valor para el argumento de la función `SayHello`.

La solución al problema planteado por el programa cliente se muestra en el siguiente ejemplo.

8.1.4 Hola Mundo con una expresión como argumento

El código de la classfactory es la siguiente:

```
public factory class HelloWorld{
public:
	static exp string^ SayHello(exp string^ argument){
		return writecode( "Hello World" + $argument );
	}
}
```

El código del cliente:

```
public class App{
public:
	static void Main(string^[] args){
		Console::WriteLine( HelloWorld::SayHello( " from Argentina." ) );
	}
}
```

La única diferencia con la classfactory anterior es la palabra reservada “exp” en la declaración del parámetro de la función SayHello:

```
	static exp string^ SayHello(exp string^ argument){
		return writecode( "Hello World" + $argument );
	}
```

Dicho cambio en la declaración del parámetro nos permite tomar como argumento expresiones sin evaluar completamente, luego podemos utilizarlas como objetos del tipo XplExpression o en expresiones writecode para generar código en memoria. Ahora el programa cliente anterior será correcto:

```
public class App{
public:
	static void Main(string^[] args){
		Console::WriteLine( HelloWorld::SayHello( " from Argentina." + args[0] )
);
	}
}
```

El resultado sería igual a escribir el siguiente código:

```
public class App{
public:
	static void Main(string^[] args){
		Console::WriteLine( "Hello World" + " from Argentina." + args[0] ) );
	}
}
```

Debe observarse que se retorna una expresión completa y no un valor. Luego, al ejecutar el programa cliente una vez compilado la expresión resultante podrá evaluarse normalmente. Puede comprobarse el resultado cambiando la classfactory a interactiva y compilando en modo interactivo con el compilador Zoe.

8.1.5 Pasando bloques como argumento

Las funciones miembros de classfactorys pueden recibir como argumento bloques de instrucciones desde el código cliente. Para esto basta con declarar el último parámetro de una función con el tipo “block”. Luego la variable de tipo “block” en la función puede ser tratada como una variable de tipo XplFunctionBody^.

El código de la classfactory es la siguiente:

```
public factory class MyTools{
public:
	static XplNode^ Repeat(exp int^ repeatCount, block argumentBlock){
		return writecode{
			for( int n=0 ; n < $repeatCount ; n++ )
				$argumentBlock;
		};
	}
}
```

El código del cliente:

```
public class App{
public:
	static void Main(string^[] args){
		MyTools::Repeat(10){
```

```
                        Console::WriteLine( "Hola Mundo." );
                };
        }
}
```

Tanto el código de la classfactory como el código cliente poseen particularidades que es preciso describir:

```
public factory class MyTools{
public:
        static XplNode^ Repeat(exp int^ repeatCount, block argumentBlock){
                return writecode{
                        for( int n=0 ; n < $repeatCount ; n++ )
                                $argumentBlock;
                };
        }
}
```

El parámetro `"argumentBlock"` se declara en último lugar, esto es preciso hacerlo así o de lo contrario es un error, tampoco puede usarse un argumento de tipo `"XplFunctionBody^"` ya que aunque dicho tipo sea internamente equivalente en dicho caso se interpretará que se desea tomar como argumento una referencia a un valor de dicho tipo y no un bloque de instrucciones per se. La función `Repeat` retorna en este caso una referencia a `XplNode` - la clase base de todos los nodos dentro del `LayerD::CodeDOM` – esto es porque en este caso no queremos retornar una expresión sino algún otro tipo de código, en éste caso un bloque de instrucciones. Finalmente las variables de tipo `"block"` y `"XplFunctionBody^"` pueden utilizarse dentro de expresiones `writecode` como argumentos del template utilizando `"$"` al igual que las variables de tipo expresión.

En el cliente remarcamos como se invoca a una función que toma como último argumento un bloque de instrucciones:

```
public class App{
public:
        static void Main(string^[] args){
                MyTools::Repeat(10){
                        Console::WriteLine( "Hola Mundo." );
                };
        }
}
```

Para pasar un bloque como argumento desde el código cliente simplemente debemos escribir las instrucciones entre llaves. Note el punto y coma al final de la llave, ello indica que `"Repeat"` es una función de una classfactory y no una instrucción nativa del lenguaje Meta D++. Es recomendable probar éste ejemplo en modo interactivo para ver el resultado más claramente.

8.1.6 Mejora de la función Repeat

La función `Repeat` del ejemplo anterior funciona, sin embargo posee una limitación que es preciso aclarar y en la práctica corregir.

Supongamos el siguiente programa cliente:

```
public class App{
public:
        static void Main(string^[] args){
                MyTools::Repeat(10){
                        Console::WriteLine( "Hola Mundo." );
                        MyTools::Repeat(5){
                                Console::WriteLine( "Inner Loop." );
                        };
                };
        }
}
```

El programa anterior – por el funcionamiento de la classfactory `MyTools` – es equivalente al siguiente programa:

```
public class App{
public:
	static void Main(string^[] args){
		for( int n = 0 ; n < 10 ; n++ ){
			Console::WriteLine( "Hola Mundo." );
			for( int n = 0 ; n < 5 ; n++ ){
				Console::WriteLine( "Inner Loop." );
			}
		}
	}
}
```

En este caso el error es evidente, la classfactory siempre retorna el ciclo for usando la variable de iteración `"n"` lo cual causará un error de compilación. Por tanto muchas veces necesitamos generar nombres de identificadores aleatorios y únicos que podamos usar en expresiones `writecode` para la transformación/generación de código.

Para obtener identificadores únicos podemos usar dos métodos:

- El tipo `XplIName`
- El prefijo `$$` en identificadores

Utilizando el tipo `XplIName` el problema del ejemplo anterior se soluciona de la siguiente forma (se resaltaron los cambios):

```
public factory class MyTools{
public:
	static XplNode^ Repeat(exp int^ repeatCount, block argumentBlock){
		XplIName^ myID = new XplIName();
		return writecode{
			for( int $myID=0 ; $myID < $repeatCount ; $myID++ )
				$argumentBlock;
		};
	}
}
```

Por tanto es preciso crear una instancia de `XplIName` usando el constructor sin argumentos para generar un nuevo identificador único. Usando el prefijo `$$` en identificadores se soluciona como sigue:

NOTA: El modificador $$ en identificadores no está implementado en la distribución del compilador Zoe que acompaña él presente documento. Por tanto no podrá utilizarse dicha funcionalidad.

```
public factory class MyTools{
public:
	static XplNode^ Repeat(exp int^ repeatCount, block argumentBlock){
		return writecode{
			for( int $$myID=0 ; $$myID < $repeatCount ; $$myID++ )
				$argumentBlock;
		};
	}
}
```

Ambos métodos dan exactamente el mismo resultado, el prefijo `$$` reutiliza el identificador generado cuando se repite el identificador que sigue al los signos `$$`. Ejemplo:

```
public factory class MyTools{
public:
	static XplNode^ Repeat(exp int^ repeatCount, block argumentBlock){
		return writecode{
			for( int $$myID=0 ; $$myID < $repeatCount ; $$myID++ )
				for( int $$otherID=0 ; $$otherID < $repeatCount ;
$$otherID++ )
```

```
                                    $argumentBlock;
                };
        }
}
```

En cualquiera de las dos soluciones posibles mostradas anteriormente se utilizará un nuevo identificador en cada llamada a la función `Repeat`. Experimente con el modo interactivo para comprender como funciona internamente éste mecanismo.

8.1.7 Retornar una clase desde una classfactory

El código de la classfactory es la siguiente:

```
public factory class MyTools{
public:
        static XplNode^ MakeClass(){
                return writecode{
                        public class TestClass{
                        public:
                                int DoSomething(){
                                        return 5;
                                }
                        }
                };
        }
}
```

El código del cliente:

```
namespace Test {
        MyTools::MakeClass();
        public class App{
        public:
                static void Main(string^[] args){
                        TestClass^ var = new TestClass();
                        Console::WriteLine( var.DoSomething().ToString() );
                }
        }
}
```

Para remarcar en el código de la classfactory tenemos los siguientes:

```
public factory class MyTools{
public:
        static XplNode^ MakeClass(){
                return writecode{
                        public class TestClass{
                        public:
                                int DoSomething(){
                                        return 5;
                                }
                        }
                };
        }
}
```

En este caso la expresión `writecode` genera el código de una clase completa y lo retorna. Al valor retornado por `writecode` también podríamos guardarlo en una variable de tipo `XplClass^` en lugar de directamente retornarlo. La función retorna un `XplNode^` pero también podría retornar directamente un `XplClass^`.

El código cliente muestra una particularidad importante de las llamadas a classfactorys:

```
namespace Test {
        MyTools::MakeClass();
        public class App{
        public:
                static void Main(string^[] args){
                        TestClass^ var = new TestClass();
                        Console::WriteLine( var.DoSomething().ToString() );
```

```
		}
	}
}
```

Las llamadas a classfactorys en programas clientes pueden encontrarse fuera del cuerpo de funciones, esto incluye cuerpo de clases, cuerpo de espacio de nombres y directamente en el cuerpo del programa.

8.1.8 Mejora de la inyección de una clase:

Se mejorará el ejemplo anterior. El código de la classfactory es la siguiente:

```
public factory class MyTools{
public:
	static XplNode^ MakeClass(){
		if(context.CurrentBlock!=null ||
		  context.CurrentNamespace==null){
			compiler.Errors.Add(
				new Error ( "No puede utilizar MakeClass fuera del cuerpo
de una clase o espacio de nombres." , context )
				);
			return null;
		}
		return writecode{
			public class TestClass{
			public:
				int DoSomething(){
					return 5;
				}
			}
		};
	}
}
```

El código del cliente quedará igual que al anterior por ahora. Los cambios que necesitan explicación en la classfactory son los siguientes:

```
public factory class MyTools{
public:
	static XplNode^ MakeClass(){
		if(context.CurrentBlock!=null ||
		  context.CurrentNamespace==null){
			compiler.Errors.Add(
				new Error ( "No puede utilizar MakeClass fuera del cuerpo
de una clase o espacio de nombres." , context )
				);
			return null;
		}
		return writecode{
			public class TestClass{
			public:
				int DoSomething(){
					return 5;
				}
			}
		};
	}
}
```

El bloque `if` lo que hace es emitir un error cuando la función `MakeClass` es llamada fuera de un cuerpo de una clase o un espacio de nombres, esto es porque no es posible insertar una clase dentro de una función o dentro del cuerpo del programa fuera de un espacio de nombres. Por tanto, el bloque `if` garantiza que la función `MakeClass` sea utilizada en el contexto adecuado.
La llamada a `"compiler.Errors.Add"` utiliza el objeto especial `"compiler"` – disponible en classfactorys – para agregar un error de compilación al programa cliente actual cuando la

función `MakeClass` no es utilizada de forma adecuada. El primer argumento del constructor de `Error` es un `string` con el mensaje de error, el segundo argumento debe ser una instancia de `XplNode` o una clase derivada que indicará el nodo del programa dentro del cual se produjo el error, en este caso pasamos el nodo `context` indicando que el error está en la llamada a función `MakeClass` en el cliente. El objeto `compiler` permite examinar los errores actuales, agregar nuevos errores como en el ejemplo, advertencias o notificaciones, también podemos consultar los tipos evaluados en el último análisis semántico ejecutado antes del tiempo de compilación actual.

8.1.9 Tomar tipos como argumentos

En general no querremos inyectar en el código cliente clases determinadas de forma estática sino que su estructura variara de acuerdo a ciertos parámetros.
A continuación se muestra como tomar como argumento un tipo de datos en una classfactory.
El código de la classfactory es el siguiente:

```
public factory class MyTools{
public:
	static XplNode^ MakeClass(type typeArgument){
		return writecode{
			public class TestClass{
				$typeArgument back;
			public:
				$typeArgument DoSomething($typeArgument argument){
					back += argument;
					return back;
				}
			}
		};
	}
}
```

El código del cliente:

```
namespace Test {
	MyTools::MakeClass( gettype(int) );
	public class App{
	public:
		static void Main(string^[] args){
			TestClass^ var = new TestClass();
			Console::WriteLine( var.DoSomething(15).ToString() );
			Console::WriteLine( var.DoSomething(12).ToString() );
		}
	}
}
```

En la classfactory sobresalen los cambios siguientes:

```
public factory class MyTools{
public:
	static XplNode^ MakeClass(type typeArgument){
		return writecode{
			public class TestClass{
				$typeArgument back;
			public:
				$typeArgument DoSomething($typeArgument argument){
					back += argument;
					return back;
				}
			}
		};
	}
```

```
}
```

El tipo especial `"type"` indica que la función toma como argumento un tipo, luego lo podemos usar como un tipo en expresiones `writecode` utilizando el prefijo `"$"` al igual que con identificadores, expresiones y bloques. Los parámetros de tipo `"type"` en tiempo de ejecución de la classfactory son equivalentes al tipo `XplType^`.
En el cliente utilizamos la expresión `"gettype"` para pasar como argumento cualquier tipo:

```
        MyTools::MakeClass( gettype(int) );
```

Con los parámetros tipo `type` es posible implementar tipos y funciones genéricas en LayerD de forma independiente al entorno de ejecución y la plataforma que usemos para el despliegue de nuestro software.

8.1.10 Retornar miembros de clase

El ejemplo muestra como retornar un miembro de clase desde una classfactory. El código de la classfactory es el siguiente:

```
public factory class MyTools{
public:
        static XplNode^ MakeFunction(type typeArgument){
                return writecode{%
                        $typeArgument DoSomething($typeArgument argument){
                                return argument + argument;
                        }
                %}.Children().FirstNode();
        }
}
```

El código del cliente:

```
namespace Test {
        public class App{
                MyTools::MakeFunction( gettype(int) );
        public:
                static void Main(string^[] args){
                        TestClass^ var = new TestClass();
                        Console::WriteLine( var.DoSomething(15).ToString() );
                }
        }
}
```

9 INFORMACIÓN EN LA WEB A CERCA DE LAYERD

- ✓ http://layerd.net
- ✓ http://layerd.blogspot.com