Programación Avanzada

Referencias Circulares
y
Namespaces

Índice

1. Introducción

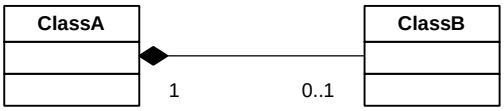
El objetivo de este documento es servir de guía para empezar a familiarizarse con la problemática de las referencias circulares y los ***namespaces*** en el lenguaje C++ [2][3]. Un pre-requisito de este documento es haber leído el documento "***Instructivo de Compilación***" [1] el cual apunta a ilustrar cómo se usan las herramientas de desarrollo.

Básicamente, este instructivo consta de una serie de ejemplos y algún ejercicio sencillo a ser realizado en máquina.

2. Referencias Circulares

Ejercicio 1

Considere el siguiente diagrama de clases:



Observación: La navegabilidad es doble.

Una implementación directa de dicho diagrama de clases es:

ClassA.hh

```
#ifndef _CLASSA_HH_
#define _CLASSA_HH_

#include "ClassB.hh"

class ClassA{
private:
  ClassB * myB;

  ...
};

#endif
```

ClassB.hh

```
#ifndef _CLASSB_HH_
#define _CLASSB_HH_

#include "ClassA.hh"

class ClassB{
private:
  ClassA * myA;

  ...
};

#endif
```

Se pide: Completar la implementación con un código simple, construir un ***makefile*** e intentar compilar. Si le da error, continúe con el ejercicio 2.

Ejercicio 2

Cuando se intente compilar las clases del ejercicio 1 se va a encontrar el problema de las referencias circulares. El hecho de que la clase ***ClassA*** necesite a la clase ***ClassB*** para ser compilada y viceversa hace que el compilador reporte el error y no pueda concluir su trabajo.
Una forma de arreglar este problema es utilizar declaraciones en avanzada (forward declarations).

ClassA.hh

```
#ifndef _CLASSA_HH_
#define _CLASSA_HH_

#include "ClassB.hh"

class ClassA{
private:
  ClassB * myB;

  ...
};

#endif
```

ClassB.hh

```
#ifndef _CLASSB_HH_
#define _CLASSB_HH_

// Declaración en avanzada de la clase ClassB
class ClassA;

class ClassB{
private:
  ClassA * myA;

  ...
};

#include "ClassA.hh"

#endif
```

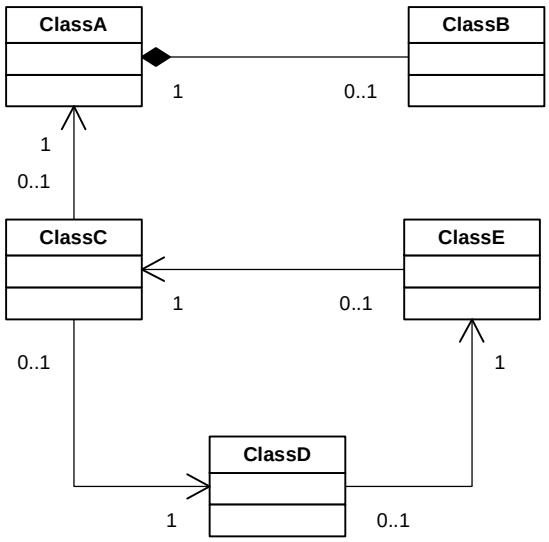
De esta manera el compilador es capaz de compilar la clase ***ClassB*** sin conocer la definición de la clase ***ClassA***. Por lo tanto se corta el ciclo y la compilación podrá proceder sin problemas.

Observación: La compilación se debe realizar de la manera normal.

Se pide: Rehacer la implementación del ejercicio 1 utilizando el mecanismo antes explicado para cortar el ciclo de dependencias. Compilar y ejecutar.

Ejercicio 3

Implemente el siguiente diagrama de clases:



Observación: La navegabilidad es doble entre ClassA y ClassB
Observación: para poder compilar esto se deben cortar todos y cada uno de los ciclos de dependencias existentes.

3. Namespaces

Introducción

Considere las siguientes clases que representan coordenadas binarias y ternarias:

Coordenada.hh

```
#ifndef _COORDENADAB_HH_
#define _COORDENADAB_HH_

class Coordenada {
private:
  int x;
  int y;

public:
  Coordenada(int x = 0, int y = 0);

  Int getX();
  Int getY();

  ...
};

#endif
```

Coordenada.hh

```
#ifndef _COORDENADAT_HH_
#define _COORDENADAT_HH_

class Coordenada {
private:
  int x;
  int y;
  int z;

public:
  Coordenada(int x = 0, int y = 0, int z = 0);

  Int getX();
  Int getY();
  Int getZ();

  ...
};

#endif
```

Ahora considere el siguiente main que transforma una coordenada binaria a una coordenada ternaria.

main.cc

```
#include <iostream>
#include "Coordenada.hh"

using namespace std;

int main() {
  int x, y;

  cout << "Ingrese la coordenada x = ";
  cin >> x;

  cout << "Ingrese la coordenada y = ";
  cin >> y;

  Coordenada binC(x, y);
  cout << "Ud. ha ingresado una coordenada binaria = (" << binC.getX() << ", "
       << binC.getY() << ")" << endl;

  // Se pasa la coordenada binaria leida a una coordenada ternaria.
  Coordenada ternC(x, y, 0);
  cout << "La coordenada ingresada transformada a ternaria = (" << ternC.getX() << ", "
       << ternC.getY() << ", " << ternC.getZ() << ")" << endl;

  return 0;
}
```

Es claro que esto acarreará un problema de conflictos de nombres. Se está haciendo referencia a dos clases diferentes, coordenadas ternarias y binarias, con el mismo nombre: ***Coordenada***. Debido a esto, el compilador no podrá decidir a qué se está queriendo referenciar cuando encuentre dicho nombre y por lo tanto no podrá compilar este ejemplo tal cual fue presentado.

Para evitar este tipo de conflictos de nombres surgen los ***namespaces***. Como se vio arriba el problema se da cuando desde un mismo archivo de código (.hh o .cc) se necesita utilizar dos

clases diferentes que se llaman de la misma forma. En C++, la forma más elegante de subsanar este problema es la utilización de ***namespaces***. La palabra reservada namespace coloca los nombres de sus miembros en una porción particular (identificada por su nombre) del espacio de nombres.

A continuación se tratarán algunos detalles sintácticos y semánticos de los ***namespaces*** y luego se verá la solución al ejemplo recién planteado en el Ejercicio 1.

Creación de ***namespaces***

La sintaxis para la creación de un ***namespace*** es la siguiente:

```
namespace namespaceId {

   // Declaraciones...
}
```

Esto produce un nuevo ***namespace*** conteniendo las declaraciones hechas dentro de él. La definición de un ***namespace*** solamente puede aparecer a nivel global, es decir, no puede estar dentro de la definición una clase o función, pero pueden definirse ***namespaces*** anidados. Además dicha definición puede extenderse a múltiples archivos simplemente repitiendo el nombre del ***namespace***:

ClassA.hh

```
namespace namespaceId {
     class ClassA { /* ... */ };
}
```

ClassB.hh

```
namespace namespaceId {
     class ClassB { /* ... */ };
}
```

Tanto ***ClassA*** como ***ClassB*** pertenecerán al mismo espacio de nombres ***namespaceId***.

Utilización de ***namespaces***

Hay dos formas de referenciar a los ***namespaces***: utilizando el ***scope resolution operator*** o utilizando la palabra clave using.
Supongamos que tenemos el siguiente ***namespace*** declarado para estudiar como funcionan ambos mecanismos:

```
namespace nsId {
     class ClassA {
          public:
               ClassA();
               void f();
     };
}
```

UTILIZANDO EL OPERADOR ::

```
int main() {

     // Cada vez que se quiera utilizar algún elemento del namespace se debera
     // especificar el nombre totalmente calificado del mismo.
     nsId::ClassA * a = new nsId::ClassA();
     a->f();

     return 1;
}
```

UTILIZANDO LA PALABRA RESERVADA USING

Incluyendo todo el espacio de nombres (forma general):

```
using namespace nsId;

int main() {
     ClassA * a = new ClassA();
     a->f();

     return 1;
}
```

```
int main() {

     using namespace nsId;

     ClassA * a = new ClassA();
     a->f();

     return 1;
}
```

Incluyendo solamente lo que se va a utilizar (forma más específica):

```
using namespace nsId::ClassA;

int main() {
     ClassA * a = new ClassA();
     a->f();

     return 1;
}
```

```
int main() {

     using namespace nsId::ClassA;

     ClassA * a = new ClassA();
     a->f();

     return 1;
}
```

Por más información acerca de los espacios de nombres en C++ referirse a [3].

El ***namespace*** de la biblioteca estándar de C++

La biblioteca estándar de C++ se encuentra en el espacio de nombres llamado ***std***. Por este motivo, cada vez que se quiera utilizar cualquiera de las funcionalidades de la misma, además de incluir los archivos correspondientes se deberá utilizar alguno de los mecanismos vistos en la sección anterior para acceder a dicho ***namespace***.

Ejercicio 1

Considere la siguiente solución al ejemplo que se encuentra en la sección Introducción del tema Namespaces:

Coordenada.hh

```
#ifndef _COORDENADAB_HH_
#define _COORDENADAB_HH_

namespace binario {
  class Coordenada {
  private:
    int x;
    int y;
```

Coordenada.hh

```
#ifndef _COORDENADAT_HH_
#define _COORDENADAT_HH_

namespace ternario {
  class Coordenada {
  private:
    int x;
    int y;
    int z;
```

```
  public:
    Coordenada(int x = 0, int y = 0);

    int getX();
    int getY();

    ...
  };
}

#endif
```

```
  public:
    Coordenada(int x = 0, int y = 0, int z = 0);

    int getX();
    int getY();
    int getZ();

    ...
  };
}

#endif
```

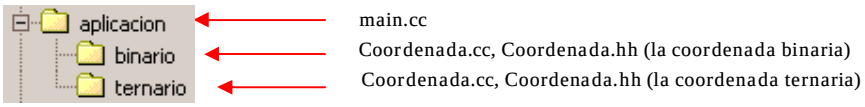
main.cc

```
#include <iostream>
#include "./binario/Coordenada.hh"
#include "./ternario/Coordenada.hh"

using namespace std;

int main() {
  int x, y;

  cout << "Ingrese la coordenada x = ";
  cin >> x;

  cout << "Ingrese la coordenada y = ";
  cin >> y;

  // Se utiliza el operador :: para explicitar a que clase se refiere.
  binario::Coordenada binC(x, y);
  cout << "Ud. ha ingresado una coordenada binaria = (" << binC.getX() << ", "
       << binC.getY() << ")" << endl;

  // Se pasa la coordenada binaria leida a una coordenada ternaria.
  // Se utiliza el operador :: para explicitar a que clase se refiere.
  ternario::Coordenada ternC(x, y, 0);
  cout << "La coordenada ingresada transformada a ternaria = (" << ternC.getX() << ", "
       << ternC.getY() << ", " << ternC.getZ() << ")" << endl;

  return 0;
}
```

Se pide:

a) Pase el ejemplo a la máquina.
b) Construya los archivos Coordenada.cc para ambas clases.
c) Construya un ***makefile*** para compilarlo y linkeditarlo.
d) Ejecute el ejemplo construido.

Observación: Notar que como los archivos que contienen las clases se llaman de igual manera se crearon dos subdirectorios, llamados de igual forma que los namespaces ***binario*** y ***ternario*** (por claridad), para almacenar dichos archivos. En definitiva, la estructura de directorios sería la siguiente:



Ejercicio 2

Compile y ejecute los siguientes ejemplos con motivo de entender cómo resuelve C++ los conflictos de nombres.

a)

```
#include <iostream>

using std::cout;
using std::endl;

namespace test {
  int i = 2;
}

class Aux {
private:
  int i;

public:
  Aux() {
    i = 1;
  }

  void accessTest() {
    cout << "El valor del atributo 'i' es: " << i << endl;

    int i = 0;

    // Podria ser la variable local o el atributo...
    cout << "El valor de 'i' es: " << i << endl;

    cout << "El valor del atributo 'i' es: " << this->i << endl;
    cout << "El valor de la variable 'i' del namespace test: " << test::i << endl;

    using namespace test;

    // Podria ser la variable del namespace, el atributo o la variable local...
    cout << "El valor de 'i' es: " << i << endl;
  }
};

int main() {
    Aux *a = new Aux();
    a->accessTest();
}
```

b)

Cambie la línea “`using namespace test;`” por la línea “`using test::i;`”
Explique el comportamiento observado.

4. Referencias

[1]	Instructivo de Compilación. Página del curso de Programación Avanzada
[2]	Tutorial de C++. www.cplusplus.com/doc/tutorial/index.html
[3]	Thinking In C++. http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html