Departamento de
Informática e Ingeniería
de Sistemas
Universidad Zaragoza

17 de junio de 2020
Examen de **Tecnología de Programación**
Grado en Ingeniería Informática
Duración total del examen: **1 horas 30 minutos**

NOTA RECORDATORIA. La evaluación final consta de las siguientes pruebas:

Prueba escrita → 60% de la nota (mínimo 5 para aprobar)
Prácticas → 40% de la nota

Ejercicio 1 [3 puntos]

Ejercicio para copiar y pegar imágenes en los enunciados de moodle:

El siguiente programa en C++ se compila con el comando `g++ main.cc -std=c++11` (gcc versión $>$ 4.8.1). Ignora errores tipográficos y potenciales memory leaks. ¿Cómo se comportaría al compilar? ¿Por qué no compilaría? Si compila, ¿qué sacaría por pantalla?

foo.h

```
class Foo {
protected:
 int i;

public:
 Foo(int i_):
   i(i_) {}

 virtual void sil()
 {  i*=2;    }

 int meta() const
 {  return i;  }
};
```

bar.h

```
#include "foo.h"
class Bar: public Foo
{
public:
 Bar() :
   Foo(-1) { }

 void sil() override
 {  i+=2;    }

 int meta() const
 {  return i-1; }

};
```

main.cc

```
#include <iostream>
#include "bar.h"
using namespace std;

void mira(Foo* foo) {
   foo->sil();
   cout<<foo->meta();
   cout<<endl;
}

int main() {
  mira(new Foo(1));
  mira(new Bar());
}
```

foo.h

```
class Foo {
protected:
 int i;

public:
 Foo(int i_):
   i(i_) {}

 void sil()
 {  i*=2;    }

 int meta() const
 {  return i;  }
};
```

bar.h

```
#include "foo.h"
class Bar: public Foo
{
public:
 Bar() :
   Foo(-1) { }

 void sil()
 {  i+=2;    }

 int meta() const
 {  return i-1; }

};
```

main.cc

```
#include <iostream>
#include "bar.h"
using namespace std;

void mira(Foo* foo) {
   foo->sil();
   cout<<foo->meta();
   cout<<endl;
}

int main() {
  mira(new Foo(1));
  mira(new Bar());
}
```

foo.h

```
class Foo {
protected:
 int i;

public:
 Foo(int i_):
   i(i_) {}

 void sil()
 {  i*=2;    }

 virtual int meta() const
 {  return i;  }
};
```

bar.h

```
#include "foo.h"
class Bar: public Foo
{
public:
 Bar() :
   Foo(-1) { }

 void sil()
 {  i+=2;    }

 int meta() const
          override
 {  return i-1; }
};
```

main.cc

```
#include <iostream>
#include "bar.h"
using namespace std;

void mira(Foo* foo) {
   foo->sil();
   cout<<foo->meta();
   cout<<endl;
}

int main() {
  mira(new Foo(1));
  mira(new Bar());
}
```

foo.h

```
class Foo {
protected:
 int i;

public:
 Foo(int i_):
   i(i_) {}

 virtual void sil()
 {  i*=2;    }

 virtual int meta() const
 {  return i;  }
};
```

bar.h

```
#include "foo.h"
class Bar: public Foo
{
public:
 Bar() :
   Foo(-1) { }

 void sil() override
 {  i+=2;    }

 int meta() const
          override
 {  return i-1; }
};
```

main.cc

```
#include <iostream>
#include "bar.h"
using namespace std;

void mira(Foo* foo) {
   foo->sil();
   cout<<foo->meta();
   cout<<endl;
}

int main() {
  mira(new Foo(1));
  mira(new Bar());
}
```

foo.h

```
class Foo {
protected:
 int i;

public:
 Foo(int i_):
   i(i_) {}

 virtual void sil()
 {  i*=2;    }

 int meta() const
 {  return i;  }
};
```

bar.h

```
#include "foo.h"
class Bar: public Foo
{
public:
 Bar() :
   Foo(-1) { }

 void sil() override
 {  i+=2;    }

 int meta() const
 {  return i-1; }

};
```

main.cc

```
#include <iostream>
#include "bar.h"
using namespace std;

void mira(Bar* bar) {
   bar->sil();
   cout<<bar->meta();
   cout<<endl;
}

int main() {
  mira(new Foo(1));
  mira(new Bar());
}
```

foo.h

```
#include <string>
using namespace std;
class Foo {
public:
 string meta() const
 {
  return
    string("FOO");
 }
};
```

bar.h

```
#include "foo.h"
class Bar :
         public Foo {
public:
 string meta() const
 {
  return
    string("BAR");
 }
};
```

main.cc

```
#include "bar.h"
#include <iostream>
void inher(Foo* foo) {
 cout<<foo->meta()<<endl;
}
template<typename F>
void templ(F* f) {
 cout<<f->meta()<<endl;
}
int main() {
 Foo foo;
 Bar bar;
 inher(&foo);
 inher(&bar);
 templ(&foo);
 templ(&bar);
}
```

foo.h

```
#include <string>
using namespace std;
class Foo {
public:
 virtual string meta()
                   const
 {
  return string("FOO");
 }
};
```

bar.h

```
#include "foo.h"
class Bar :
     public Foo {
public:
 string meta() const
     override {
   return
     string("BAR");
 }
};
```

main.cc

```
#include "bar.h"
#include <iostream>
void inher(Foo* foo) {
 cout<<foo->meta()<<endl;
}
template<typename F>
void templ(F* f) {
 cout<<f->meta()<<endl;
}
int main() {
 Foo foo;
 Bar bar;
 inher(&foo);
 inher(&bar);
 templ(&foo);
 templ(&bar);
}
```

foo.h

```
template<typename T>
class Foo {
protected:
 T t;
public:
 Foo(const T& t) :
    t(t) { }
 T meta() const {
   return t;
 }
};
```

bar.h

```
#include "foo.h"
template<typename T>
class Bar {
 Foo<T>* foo;
public:
  Bar(Foo<T>* foo):
    foo(foo) { }
  T tolo() const {
    return
       foo->meta();
  }
};
```

main.cc

```
#include "bar.h"
#include <iostream>
using namespace std;
int main() {
 Bar<int> ibar(
       new Foo<int>(42));
 cout<<ibar.tolo()<<endl;

 Bar<float> fbar(
   new Foo<float>(4.8f));
 cout<<fbar.tolo()<<endl;

 Bar<double> dbar(
     new Foo<int>(1024));
 cout<<dbar.tolo()<<endl;
}
```


foo.h

```
template<typename T>
class Foo {
protected:
 T t;
public:
 Foo(const T& t) :
    t(t) { }
 T meta() const {
   return t;
 }
};
```

bar.h

```
#include "foo.h"
template<typename T>
class Bar {
 Foo<T>* foo;
public:
  Bar(Foo<T>* foo):
    foo(foo) { }
  T tolo() const {
    return
       foo->meta();
  }
};
```

main.cc

```
#include "bar.h"
#include <iostream>
using namespace std;
int main() {
 Bar<int> ibar(
       new Foo<int>(42));
 cout<<ibar.tolo()<<endl;

 Bar<float> fbar(
   new Foo<float>(4.8f));
 cout<<fbar.tolo()<<endl;

 Bar<double> dbar(
  new Foo<double>(1024));
 cout<<dbar.tolo()<<endl;
}
```


Foo.hs

```
module Foo where
pet :: [Integer]
pet = map (^ 2) [1..]

foo :: Int -> [t] -> [[t]]
foo 0 _   = [[]]
foo _ []  = [  ]
foo n (x:xs) =
  [ x:rest | rest <- foo (n-1) xs ]
  ++ foo n xs
```

Main.hs

```
import Foo
bar :: Int -> Int -> [[Integer]]
bar n m   = foo n (take m pet)

main = do
    print $ bar 1 5
    print $ bar 3 4
```

Foo.hs

```
module Foo where
pet :: [Integer]
pet = map (^ 2) [1..]

foo :: Int -> [t] -> [[t]]
foo 0 _  = [[]]
foo _ [] = [[]]
foo n (x:xs) =
  [ x:rest | rest <- foo (n-1) xs ]
  ++ foo n xs
```

Main.hs

```
import Foo
bar :: Int -> Int -> [[Integer]]
bar n m   = foo n (take m pet)

main = do
    print $ bar 1 5
    print $ bar 3 4
```

Foo.hs

```
module Foo where
pet :: [Integer]
pet =  map (2 ^) [1..]

foo :: Int -> [t] -> [[t]]
foo 0 _  = [[]]
foo _ [] = [  ]
foo n (x:xs) =
  [ x:rest | rest <- foo (n-1) xs ]
  ++ foo n xs
```

Main.hs

```
import Foo
bar :: Int -> Int -> [[Integer]]
bar n m   = foo n (take m pet)

main = do
    print $ bar 1 5
    print $ bar 3 4
```

Foo.hs

```
module Foo where
pet :: [Integer]
pet =  map (2 ^) [1..]

foo :: Int -> [t] -> [[t]]
foo 0 _  = [[]]
foo _ [] = [[]]
foo n (x:xs) =
  [ x:rest | rest <- foo (n-1) xs ]
  ++ foo n xs
```

Main.hs

```
import Foo
bar :: Int -> Int -> [[Integer]]
bar n m   = foo n (take m pet)

main = do
    print $ bar 1 5
    print $ bar 3 4
```

Foo.hs

```
module Foo where
pet :: [Integer]
pet = map (^ 2) [1..]

foo :: Int -> [t] -> [[t]]
foo 0 _  = [[]]
foo _ [] = [  ]
foo n xs =
  [ x:rest | x<-xs, rest <- foo (n-1) xs ]
```

Main.hs

```
import Foo
bar :: Int -> Int -> [[Integer]]
bar n m   = foo n (take m pet)

main = do
    print $ bar 1 5
    print $ bar 3 4
```

Foo.hs

```
module Foo where
pet :: [Integer]
pet =  map (2 ^) [1..]

foo :: Int -> [t] -> [[t]]
foo 0 _  = [[]]
foo _ [] = [  ]
foo n xs =
  [ x:rest | x<-xs, rest <- foo (n-1) xs ]
```

Main.hs

```
import Foo
bar :: Int -> Int -> [[Integer]]
bar n m   = foo n (take m pet)

main = do
    print $ bar 1 5
    print $ bar 3 4
```

Ejercicio 2 [2 puntos]

En Programación Orientada a Objetos existen dos formas fundamentales para implementar el **polimorfismo**: la **herencia** (polimorfismo de inclusión) y la **programación genérica** (polimorfismo paramétrico). Las dos técnicas se utilizan por separado en la definición de clases, y también pueden combinarse.

En el lenguaje de tu elección (C++ o Java) escribe un **pequeño** ejemplo de código que incluya:

- Dos clases que cumplan lo siguiente:
  - Una clase base que | **no sea genérica** | **sea genérica** |.
  - Una clase derivada de la anterior, que | sea también **genérica** | **no sea genérica** |.

  En la implementación de la herencia debes reflejar el funcionamiento del polimorfismo de inclusión, y en el genérico el polimorfismo paramétrico.
- Un programa principal que demuestre con tus clases los dos tipos de polimorfismo.

Los ficheros fuente que generes se empaquetarán en un **único archivo** llamado

`ejercicio2_<NIP>.zip`

que se entregará como fichero adjunto a esta pregunta.

En el campo de texto habilitado en la pregunta, **responde brevemente** a las siguientes cuestiones sobre tu código:

(a) ¿ Dónde se refleja en tu código el uso del polimorfismo de inclusión ?
(b) ¿ Cómo cambiaría tu código si el lenguaje no soportara polimorfismo de inclusión ?
(c) ¿ Dónde se refleja en tu código el uso del polimorfismo paramétrico ?
(d) ¿ Cómo cambiaría tu código si el lenguaje no soportara polimorfismo paramétrico ?

Ejercicio 3 [2 puntos]

Una forma de estructurar el sistema de ficheros dentro de un computador es mediante un **árbol de directorios**. Cada **directorio** puede contener varios elementos, en particular **ficheros** u otros directorios (**subdirectorios**). A los elementos del árbol (directorios o ficheros) se les denomina **nodos**, y cada uno de ellos tiene un nombre (cadena de texto). El nodo a partir del que parte todo el árbol de directorios es un directorio y se le denomina directorio **raíz**. Dos nodos dentro del mismo directorio no pueden tener el mismo nombre (se distingue entre mayúsculas y minúsculas). Un fichero tiene un **tamaño** concreto (en bytes), mientras que el tamaño de un directorio es la suma de los tamaños de todos los nodos que contiene.

Para referenciar de forma textual cualquier nodo dentro de ese árbol de directorios se utiliza su *path* o **ruta**: una secuencia de directorios y subdirectorios desde el directorio raíz hasta llegar al elemento referenciado, que en este ejercicio se representará mediante una lista de cadenas de texto que llevan desde la raíz hasta el nodo específico. Sobre este tema, y utilizando programación funcional mediante el lenguaje Haskell, **se pide**:

(a) Diseña un tipo de datos `ArbolDirectorios` que represente un arbol de directorios.

(b) Diseña una función `stat` que devuelva el tamaño de un nodo del arbol indicado por una ruta. Si la ruta no existe en el arbol de directorios entonces devuelve cero.

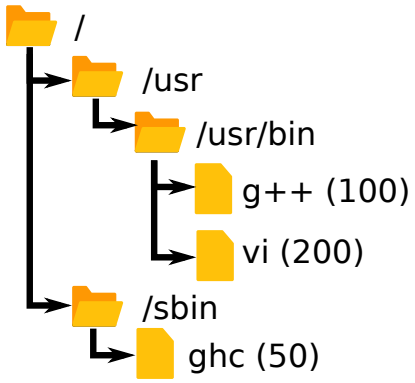
```
stat :: ArbolDirectorios->[String]->Integer
```

(c) Diseña una función `find` que devuelva una lista con todas las rutas posibles de un arbol de directorios. Se deberá incluir tanto la ruta que lleva a cada directorio como la ruta que lleva a cada uno de los subdirectorios.

```
find :: ArbolDirectorios->[[String]]
```

Ejemplo:

Arbol de directorios `ejemplo`



Resultados

```
*Main> stat ejemplo ["sbin","ghc"]
50
*Main> stat ejemplo ["usr","bin","g++"]
100
*Main> stat ejemplo ["usr","bin","vi"]
200
*Main> stat ejemplo []
350
*Main> stat ejemplo ["usr"]
300
*Main> stat ejemplo ["var","share"]
0
*Main> find ejemplo
[[],["usr"],["usr","bin"],["usr","bin","g++"],
 ["usr","bin","vi"],["sbin"],["sbin","ghc"]]
```

Se permite definir tipos de datos y funciones auxiliares adicionales si lo consideras necesario. Se valorará positivamente la **brevedad y concisión** de las soluciones propuestas. Se puede utilizar cualquier función del `Prelude` de Haskell, pero no otras funciones del lenguaje.

Ejercicio 4 [3 puntos]

Para construir un juego arcade en 2D (de los de marcianitos...) se necesita implementar un sistema de **detección de colisiones**, que permita saber si dos objetos van a chocar o no. El sistema general más sencillo para ello consiste en tener una representación poligonal de cada objeto, y ver si alguno de los vértices de un objeto está en el interior del otro y viceversa. En un juego podremos tiener distintos tipos de objetos, y se debe poder comprobar las colisiones entre todos los tipos. Para evitar la explosión exponencial de funciones podemos emplear la herencia y la programación genérica.

En este ejercicio deberás implementar, en el lenguaje orientado a objetos de tu elección (C++ o Java), un sistema sencillo de detección de colisiones basado en ese principio.

Para ello, implementa las siguientes clases:

- Una clase **Point**, que se creará a partir de sus coordenadas reales (x,y). Puedes ir añadiendo a esta clase todos los métodos y operadores que creas necesarios para la realización del ejercico.

- Distintas clases que nos permitan representar objetos:
  - **Circle**: círculo definido a partir del centro y el radio. Cuando sea necesaria su representacion poligonal, será una lista de 32 puntos.
  - **Rectangle**: rectángulo definido a partir de dos puntos, que representan las esquinas inferior izquierda y superior derecha. Su representación poligonal estará formada por 4 puntos (las 4 esquinas del rectángulo).
  - **Polygon**: polígono definido directamente mediante su lista de puntos.

  Todos los objetos deben ofrecer cierta funcionalidad común:

  - Devolver su lista de puntos:

    ```
    ???? points()
    ```

  - Averiguar si contiene a un punto dado:

    ```
    bool contains(??? Point x)
    ```

  Cada objeto puede implementarlas de la forma más adecuada.
  Para un polígono, el test para ver si contiene a un punto puede aproximarse mediante una caja rectangular que contenga todos los puntos del polígono, definida mediante el máximo y mínimo de las coordenadas de los puntos, en ambos ejes (x e y).

- Define una función que compruebe si dos objetos cualesquiera colisionan:

  ```
  bool collide(??? a, ??? b)
  ```

  Dos objetos colisionan si aguno de los puntos del primero está contenido en el segundo, o bien alguno de los puntos del segundo está contenido en el primero.

- Define una función que compruebe si un objeto colisiona con cualquiera de una lista de otros objetos **cualesquiera** (genéricos o de un tipo determinado):

  ```
  bool collide_any(??? objeto, ??? lista)
  ```

  La función tiene que operar tanto con listas de objetos genéricos, que contengan objetos variados (círculos, rectángulos...), como con listas homogéneas de uno de esos tipos.

- Entre ciertos tipos de objetos es más eficiente detectar si colisionan entre sí. Define versiones optimizadas de la función `collide()` para dos objetos, para comprobar:
  - la colisión entre dos círculos
  - la colisión entre dos rectángulos

  Asegurate de que la función `collide_any()` utiliza la versión optimizada en el caso de que sus parámetros coincidan con esos tipos.

Escribe un pequeño programa principal que demuestre la funcionalidad implementada.

Puedes utilizar toda la funcionalidad definida en la biblioteca estándar de C++14 o Java.