Universidad de Oviedo

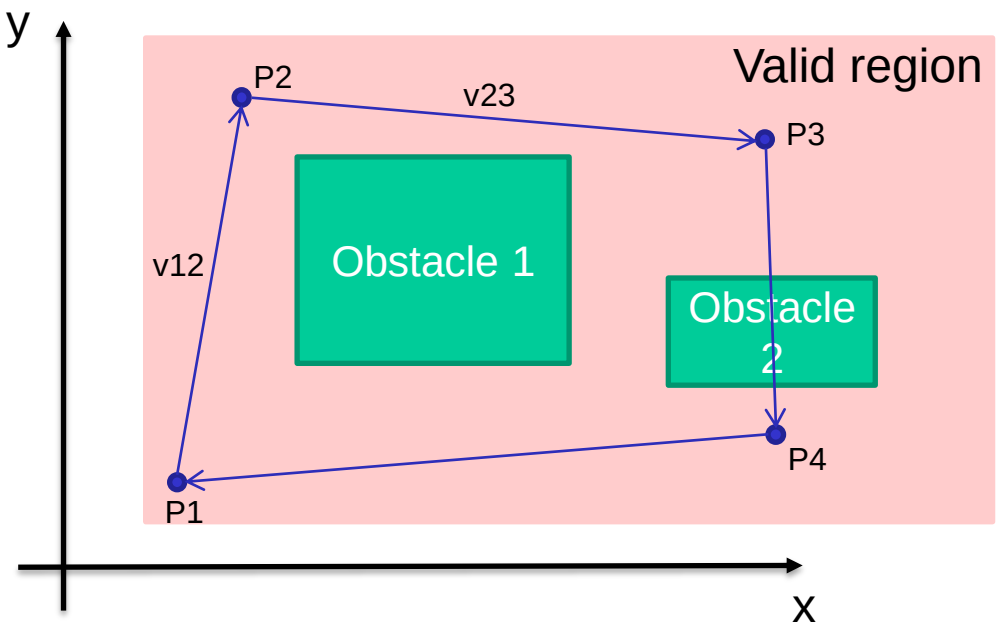
Ejemplo básico de C++
Trayectorias en 2D

Ignacio Alvarez García

Sept - 2024

Ejemplo base C++: trayectorias 2D

- Muchos programas (para control de dispositivos mecatrónicos y muchos otros) necesitarán manejar los mismos conceptos



- De la figura surgen las abstracciones (clases):
  - Punto en el plano xy
  - Vector entre 2 puntos
  - Rectángulo en el plano xy
  - Trayectoria: conjunto ordenado de puntos + rectángulo de límites

Ejemplo base C++: clases

- Cada concepto se encapsula en una clase, en forma de :
  - **Variables miembro**: datos que definen las propiedades y el estado actual de cada objeto de esta clase
  - **Funciones miembro**: código que permite interactuar con los objetos de la clase
  - Una 1ª aproximación:

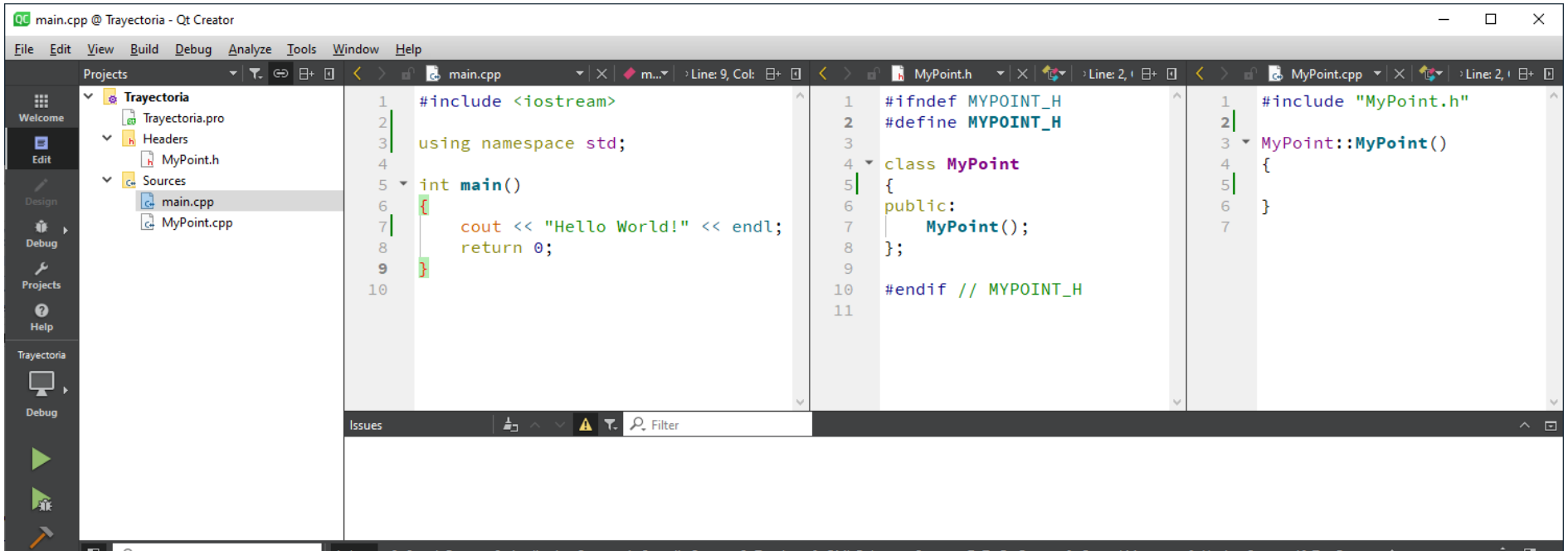
Clase	Vbles miembro		Funciones miembro	
Point	x y name	reales string	SetXY(i_x,i_y) GetName() …	Da valor a x e y Devuelve nombre …
Vector	vx vy	reales	FromPoints(pt1,pt2) GetLength() …	Dife entre 2 puntos Devuelve longitud …
Trajectory	points closed	Lista de Point bool	GetLength() Optimize() …	Devuelve distancia total Reordena puntos …
Rectangle	tl br name	Point string	SetPoints(i_tl,it_br) GetWidth() …	Da valor a tl,br Devuelve ancho …

Ejemplo base C++: clases

- Crear una proyecto en Qt-Creator:
  - File → New Project → (Non-Qt Project , Plain C++ App)
  - Name & Path (no usar á é ö , : () ñ space …)
  - Build system: qmake
  - Kit: Desktop … MinGW-64bit
- Crear una clase para el proyecto:
  - File → New File → (C/C++ , C++ class)
  - Class name: MyPoint
  - Add to Project

Ejemplo base C++: clases

- Resultado tras añadir clase MyPoint:
  - Proyecto con 4 archivos (.pro, .cpp, .h)
  - Archivos .cpp : funciones con código a ejecutar
  - Archivos .h : declaraciones
  - Archivo .pro : configuración para compilación (se necesita modificar → CONFIG **+**= qt)



Ejemplo base C++: clases

- Rellenando MyPoint de forma sencilla (y no muy correcta):
  - En MyPoint.h , bajo public: , declarar vbles miembro con su tipo
    - Una vble miembro puede ser de tipo básico (int ,float, array, struct, …)
    - Una vble miembro puede ser de tipo clase: se necesita incluir el .h que declara dicha clase

QString: clase ya realizada en Qt-SDK que facilita manipular cadenas de caracteres

MyPoint.h

```
1   #ifndef MYPOINT_H
2   #define MYPOINT_H
3
4   #include <QString>
5
6   class MyPoint
7   {
8   public:
9       float x,y;
10      QString name;
11      MyPoint();
12  };
13
14  #endif // MYPOINT_H
15
```

Ejemplo base C++: clases

- Utilizando MyPoint de forma sencilla (y no muy correcta):
  - En main.cpp , declarar y utilizar objetos de tipo MyPoint
    - Se necesita incluir el .h que declara la clase
    - Los objetos serán variables locales de main
    - Se accede a los miembros del objeto con el operador punto
    - Los objetos ocupan posiciones de memoria diferentes

main.cpp | main() -> int

```
1  #include <iostream>
2  #include "MyPoint.h"
3
4  int main()
5  {
6      MyPoint pt1,pt2;
7
8      pt1.x=2;
9      pt1.y=20.5;
10     pt1.name="PT1";
11
12     pt2.x=6;
13     pt2.y=8;
14     pt2.name="PT2";
15
16     return 0;
17 }
18
```

Memoria (stack)

pt1	x	2
	y	4
	name	"PT1"
pt2	x	6
	y	20.5
	name	"PT2"

Utilizar Debugger para ver contenidos de variables

Ejemplo base C++: clases

- Crear y rellenar MyVector de forma similar:
  - Variables miembro: vx, vy (float)
  - No necesita nombre
  - Añadir objeto MyVector v en main
  - Calcular v como diferencia pt2-pt1 en main

main.cpp | main() -> int | Line: 13, Col: 13

```
1  #include <iostream>
2  #include "MyPoint.h"
3  #include "MyVector.h"
4
5  int main()
6  {
7      MyPoint pt1,pt2;
8
9      pt1.x=2;
10     pt1.y=4;
11     pt1.name="PT1";
12
13     pt2.x=6;
14     pt2.y=20.5;
15     pt2.name="PT2";
16
17     MyVector v;
18     v.vx=pt2.x-pt1.x;
19     v.vy=pt2.y-pt1.y;
20
21     return 0;
22 }
23
```

Memoria (stack)

pt1	x	2
	y	4
	name	"PT1"
pt2	x	6
	y	20.5
	name	"PT2"
v	vx	4
	vy	16.5

Ejemplo base C++: clases

- Mejorando el código (paso 1):
  - Un vector como diferencia de 2 puntos se calcula siempre igual: es una funcionalidad común a todos los vectores
  - Añadimos dicha funcionalidad a MyVector para asignar vx vy a partir de dos puntos:
    - Bajo public: en MyVector.h crear función con dos argumentos tipo MyPoint y que retorna void (incluir "MyPoint.h")
    - Seleccionar función, pulsar botón derecho → Refractor → Add definition in MyVector.cpp
    - Rellenar función: dentro de una función miembro de una clase, se referencia a las variables (y otras funciones) miembro de la misma clase directamente, sin punto
    - Cambiar main para utilizar esta función

MyVector.h

```
1  #ifndef MYVECTOR_H
2  #define MYVECTOR_H
3
4  #include "MyPoint.h"
5
6  class MyVector
7  {
8  public:
9      float vx,vy;
10     MyVector();
11     void FromPoints(MyPoint from,MyPoint to);
12 };
13
14 #endif // MYVECTOR_H
15
```

MyVector.cpp

```
8  void MyVector::FromPoints(MyPoint from, MyPoint to)
9  {
10     vx=to.x-from.x;
11     vy=to.y-from.y;
12 }
13
```

main.cpp

```
2  #include "MyPoint.h"
3  #include "MyVector.h"
16
17     MyVector v;
18     v.FromPoints(pt1,pt2);
```

Ejemplo base C++: clases

- Mejorando el código (paso 2):
  - La longitud de un vector se calcula siempre igual: es una funcionalidad común a todos los vectores
  - Añadimos dicha funcionalidad a MyVector para obtener la longitud a partir de vx vy:
    - Bajo public: en MyVector.h crear función sin argumentos y que retorna float (función const si no modifica el objeto)
    - Seleccionar función, pulsar botón derecho → Refractor → Add definition in MyVector.cpp
    - Rellenar función (se pueden utilizar variables locales para uso temporal) (habrá que incluir <math.h> para usar sqrt)
    - Cambiar main para utilizar esta función

MyVector.h

```
12      float GetLength() const;
```

MyVector.cpp

```
2   #include <math.h>

15  float MyVector::GetLength() const
16  {
17      float len;
18      len=sqrt(vx*vx+vy*vy);
19      return len;
20  }
```

main.cpp

```
20      float len;
21      len=v.GetLength();
```

Ejemplo base C++: clases

- Mejorando el código (paso 3):
  - Cualquier función (por ejemplo main) podría cambiar posteriormente v.vx ó v.vy , "destrozando" el cálculo
    - Si no queremos permitirlo, vx vy deben ser declaradas bajo private: en lugar de bajo public:
    - De esta manera, sólo las funciones miembro de MyVector tienen el derecho de acceder a vx vy
    - Si deseamos que otra función/clase (main) pueda obtener el valor de vx vy, pero no modificarlas, añadimos funciones tipo get
    - Las funciones con "poco" código se suelen implementar en el .h, y se preceden con inline para su ejecución rápida

MyVector.h

```
6  class MyVector
7  {
8  private:
9      float vx,vy;
10 public:
11     MyVector();
12     void FromPoints(MyPoint from,MyPoint to);
13     float GetLength() const;
14     inline float GetVX() const { return vx; }
15     inline float GetVY() const { return vy; }
16 };
```

main.cpp

```
25     float vector_x=v.GetVX();
26     float vector_y=v.vy;     'vy' is a private member of 'MyVector'…
```

Ejemplo base C++: clases

- Mejorando el código (paso 4):
  - Una función especial (constructor) es llamada automáticamente cada vez que se crea un objeto de una clase
    - En el constructor podemos establecer los valores por defecto de las variables y/o otras inicializaciones
    - Se pueden añadir varios constructores con argumentos para crear directamente objetos con los valores iniciales deseados
    - Todos los constructores se deben llamar como la clase, y no devuelven nada (ni siquiera void)
    - Por supuesto, todas las funciones de una clase pueden llamar a otras funciones de la misma clase
    - El constructor que será invocado depende de los argumentos que se indiquen en la declaración del objeto
    - Existe un constructor por defecto (oculto al programador) llamado copy constructor, que permite crear un objeto a partir de otro de la misma clase (ver creación de v3)

MyVector.h

```
10  public:
11      MyVector();
12      MyVector(float i_vx,float i_vy);
13      MyVector(MyPoint from,MyPoint to);
```

MyVector.cpp

```
4   MyVector::MyVector()
5   {
6       vx=0;
7       vy=0;
8   }

10  MyVector::MyVector(float i_vx, float i_vy)
11  {
12      vx=i_vx;
13      vy=i_vy;
14  }
15
16  MyVector::MyVector(MyPoint from, MyPoint to)
17  {
18      FromPoints(from,to);
19  }
```

main.cpp

```
17      MyVector v0,v1(3.5,2.7),v2(pt1,pt2),v3(ux);
```

Ejemplo base C++: clases

- Mejorando el código (paso 5):
  - Todas las funciones (incluidos los constructores) pueden tener argumentos por defecto
    - Los valores por defecto se declaran solamente en el .h
    - Si no se pasa algún argumento, tomará el valor por defecto
    - Esto nos permite, por ejemplo, prescindir del constructor sin argumentos, ya que es un caso por defecto del constructor con dos argumentos float

MyVector.h

```
10  public:
11      MyVector(float i_vx=0,float i_vy=0);
12      MyVector(MyPoint from,MyPoint to);
```

MyVector.cpp

```
4   MyVector::MyVector(float i_vx, float i_vy)
5   {
6       vx=i_vx;
7       vy=i_vy;
8   }
9
10  MyVector::MyVector(MyPoint from, MyPoint to)
11  {
12      FromPoints(from,to);
13  }
```

main.cpp

```
17      MyVector v0,v1(3.5,2.7),ux(1),v2(pt1,pt2),v3(ux);
```

Ejemplo base C++: clases

- Mejorando el código (paso 6):
  - Una función especial (destructor) es llamada automáticamente cada vez que se destruye un objeto de una clase
    - En el destructor podemos realizar operaciones que se asegura que el objeto se elimina de forma segura (por ejemplo, para un objeto tipo Motor, antes de eliminar asegurar que se pone velocidad cero)
    - El destructor va sin argumentos
    - El destructor se debe llamar como la clase precedido por ~, y no devuelve nada (ni siquiera void)
    - Si no existe destructor, se invoca a uno por defecto (oculto al programador) que no hace nada

MyVector.h

```
10  public:
11      MyVector(float i_vx=0,float i_vy=0);
12      MyVector(MyPoint from,MyPoint to);
13      ~MyVector();
```

MyVector.cpp

```
3   #include <iostream>
4
5   MyVector::MyVector(float i_vx, float i_vy)
6   {
7       vx=i_vx;
8       vy=i_vy;
9       std::cout << "Created MyVector with vx=" << vx <<
10                   " , vy=" << vy << "\n";
11  }
12
13  MyVector::MyVector(MyPoint from, MyPoint to)
14  {
15      FromPoints(from,to);
16      std::cout << "Created MyVector from points: vx=" << vx <<
17                   " , vy=" << vy << "\n";
18  }
19
20  MyVector::~MyVector()
21  {
22      std::cout << "MyVector destructed\n";
23  }
```

Ejemplo base C++: clases

- Ejercicio: añadir constructor(es) y destructor a MyPoint
  - Los constructores y el destructor deben escribir en pantalla los datos del punto (name, x, y)
  - Añadimos constructor copia (sobrecarga el constructor por defecto) para ver qué sucede con él
  - Para escribir name tenemos un problema: std::cout no reconoce a QString porque no es una clase standard de C++ → usamos la función toStdString() de QString , que devuelve un std::string (que ya es standard C++) equivalente

Ejemplo base C++: clases

MyPoint.h

```
1  #ifndef MYPOINT_H
2  #define MYPOINT_H
3
4  #include <QString>
5
6  class MyPoint
7  {
8  public:
9      float x,y;
10     QString name;
11     MyPoint(QString i_name,float i_x=0,float i_y=0);
12     MyPoint(const MyPoint& other);
13     ~MyPoint();
14 };
15
16 #endif // MYPOINT_H
```

MyPoint.cpp

```
1  #include "MyPoint.h"
2  #include <iostream>
3
4  MyPoint::MyPoint(QString i_name, float i_x, float i_y)
5  {
6      x=i_x;
7      y=i_y;
8      name=i_name;
9      std::cout << "Created point " << name.toStdString() << "(" << x << "," << y << ")\n";
10 }
11
12 MyPoint::MyPoint(const MyPoint &other)
13 {
14     x=other.x;
15     y=other.y;
16     name="Copy of "+other.name;
17     std::cout << "Created point " << name.toStdString() << "(" << x << "," << y << ")\n";
18 }
19
20 MyPoint::~MyPoint()
21 {
22     std::cout << "Removed point " << name.toStdString() << "\n";
23 }
```

Constructor copia:
A continuación se verá el por qué de este formato

Ejemplo base C++: clases

- Si ahora modificamos main y ejecutamos:

```
#include "MyPoint.h"
#include "MyVector.h"

int main()
{
    MyPoint pt1("PT1",2,4),pt2("PT2",6,20.5);

    MyVector v0,v1(3.5,2.7),ux(1),v2(pt1,pt2),v3(ux);

    return 0;
}
```

```
D:\Qt6\Tools\QtCreator\bin\qtcreator_process_stub.exe
Created point PT1(2,4)
Created point PT2(6,20.5)
Created MyVector with vx=0 , vy=0
Created MyVector with vx=3.5 , vy=2.7
Created MyVector with vx=1 , vy=0
Created point Copy of PT2(6,20.5)
Created point Copy of PT1(2,4)
Created point Copy of Copy of PT2(6,20.5)
Created point Copy of Copy of PT1(2,4)
Removed point Copy of Copy of PT1
Removed point Copy of Copy of PT2
Created MyVector from points: vx=4 , vy=16.5
Removed point Copy of PT1
Removed point Copy of PT2
MyVector destructed
MyVector destructed
MyVector destructed
MyVector destructed
MyVector destructed
Removed point PT2
Removed point PT1
Press <RETURN> to close this window...
```

¿Qué ha ocurrido aquí?
Los argumentos son variables nuevas, se crean en el stack (como copia de los valores que se han pasado) cuando se invoca a una función, y se destruyen cuando termina la función

```
MyVector::MyVector(MyPoint from, MyPoint to)
{
    FromPoints(from,to);
}

void MyVector::FromPoints(MyPoint from, MyPoint to)
{
```

Ejemplo base C++: clases

- Mejorando el código (paso 7):
  - Utilizar el operador referencia & al pasar argumentos de tipo clase a funciones
    - La referencia & evita que se cree y use un objeto nuevo, sino el mismo que se ha pasado como argumento
    - Dado que se usa el objeto de la función llamadora (no una copia), una función podría modificar la variable de la llamadora: se evita (si se desea) con const

MyVector.h

```
12      MyVector(const MyPoint& from,const MyPoint& to);
13      ~MyVector();
14      void FromPoints(const MyPoint& from,const MyPoint& to);
```

MyVector.cpp

```
13  MyVector::MyVector(const MyPoint& from,const MyPoint& to)
14  {
15      FromPoints(from,to);
16      std::cout << "Created MyVector from points: vx=" << vx <<
17                   " , vy=" << vy << "\n";
18  }

25  void MyVector::FromPoints(const MyPoint& from,const MyPoint& to)
26  {
27      vx=to.x-from.x;
28      vy=to.y-from.y;
29  }
```

Se puede obtener el valor from.x, pero no modificarlo desde esta función al ser const
Si se modificase, se estaría modificando el valor pt1.x de main()

```
D:\Qt6\Tools\QtCreator\bin\qtcreator_process_stub.exe
Created point PT1(2,4)
Created point PT2(6,20.5)
Created MyVector with vx=0 , vy=0
Created MyVector with vx=3.5 , vy=2.7
Created MyVector with vx=1 , vy=0
Created MyVector from points: vx=4 , vy=16.5
MyVector destructed
MyVector destructed
MyVector destructed
MyVector destructed
MyVector destructed
Removed point PT2
Removed point PT1
Press <RETURN> to close this window...
```

Ejemplo base C++: clases

- Ejercicios propuestos

1) Repetir el programa desde cero con la solución final (intentar aplicar los conocimientos, recurrir al ejemplo ya hecho sólo para consulta)
2) Utilizar editor/compilador/depurador gráfico (instrucciones de descarga y uso en http://isa.uniovi.es/~ialvarez/Curso/descargas/SimuladorWSL2/almar-simulator-install-users.pdf)

   para comprobar con más claridad lo que sucede en cada caso.

Ejemplo base C++: clases

- Ejercicios propuestos

3) Añadir clase MyRect para gestionar rectángulos (un rectángulo se define por dos puntos, nos puede interesar su ancho, alto, área, saber si un punto cae dentro del rectángulo, etc.)

4) Añadir clase MyTraj para gestionar trayectorias
   - Una trayectoria contiene un conjunto ordenado de puntos; se puede utilizar la clase QList<tipo> de Qt-Sdk
   - Una trayectoria puede contener también un rectángulo con los límites: no se admiten puntos que no estén dentro de los límites
   - De una trayectoria nos puede interesar, por ejemplo: añadir punto (si está dentro de límites) , longitud total, nº de puntos, escribir en pantalla

Se dispone de solución a continuación (intentar aplicar los conocimientos, utilizar la solución solamente a modo de referencia)
La solución incluye necesidades aún no comentadas (construcción de objetos junto a la clase que los alberga, cuando no tienen constructor por defecto)

Ejercicio 4: solución

MyTraj.h

```
1   #ifndef MYTRAJ_H
2   #define MYTRAJ_H
3
4   #include <QList>
5   #include "MyPoint.h"
6   #include "MyRect.h"
7
8   class MyTraj
9   {
10  private:
11      QList<MyPoint> points;
12      MyRect limits;
13  public:
14      MyTraj(float x_min,float x_max,float y_min,float y_max);
15      bool AddPoint(float x,float y);
16      float GetLength() const;
17      void Print() const;
18  };
19
20  #endif // MYTRAJ_H
```

Sin constructor por defecto, sólo constructor con 4 argumentos que definen los límites: para crear un objeto tipo MyTraj será siempre obligatorio especificar los límites

Ejercicio 4: solución

MyTraj.cpp

```
1  #include "MyTraj.h"
2  #include "MyVector.h"
3  #include <iostream>
4
5
6  MyTraj::MyTraj(float x_min, float x_max, float y_min, float y_max) :
7                  limits(MyPoint("Top-Left",x_min,y_min),MyPoint("Top-Right",x_max,y_max))
8  {
9
10 }
11
12 bool MyTraj::AddPoint(float x,float y)
13 {
14     QString name;
15     int curSize=points.size();
16     if (curSize==0)
17         name="Starting point";
18     else
19         name=QString("Point %1").arg(curSize+1);
20     MyPoint newPoint(name,x,y);
21     bool canAdd=limits.IsInside(newPoint);
22     if (canAdd)
23         points.append(newPoint);
24     return canAdd;
25 }
26
27 float MyTraj::GetLength() const
28 {
29     float sum;
30     int i;
31     for (i=1,sum=0;i<points.size();i++) {
32         MyVector v(points[i-1],points[i]);
33         sum+=v.GetLength();
34     }
35     return sum;
36 }
37
38 void MyTraj::Print() const {
39     std::cout << "TRAYECTORY POINTS:\n";
40     for (int i=0;i<points.size();i++) {
41         points[i].Print();
42     }
43 }
44
```

Cuando se crea un objeto de tipo MyTraj, se debe crear automáticamente el objecto (variable miembro) limits de tipo MyRect: si no tiene constructor por defecto, se debe hacer antes del cuerpo de la función (cuando se entra en las instrucciones de la función constructor de MyTraj, el objeto limits ya debe estar creado)

Ejercicio 4: solución

main.cpp

```
1  #include "MyTraj.h"
2  #include <iostream>
3
4  int main()
5  {
6      MyTraj t(10,80,20,60);
7
8      t.AddPoint(20,40);
9      t.AddPoint(20,75);
10     t.AddPoint(60,40);
11     t.AddPoint(60,60);
12
13     t.Print();
14
15     float len=t.GetLength();
16     std::cout << "Trajectory length: " << len << "\n";
17
18     return 0;
19 }
```

No será añadido por estar fuera de límites

```
D:\Qt6\Tools\QtCreator\bin\qtcreator_process_stub.exe
TRAYECTORY POINTS:
Copy of Copy of Copy of Starting point(20,40)
Copy of Copy of Point 2(60,40)
Copy of Point 3(60,60)
Trajectory length: 60
Press <RETURN> to close this window...
```

¿Qué ha ocurrido aquí?
La función append() de QList crea internamente nuevos MyPoint en la operación de añadir

Ejemplo base C++: clases

- Ejercicios propuestos

5) Añadir al programa anterior una lista de obstáculos (rectángulos) y el chequeo de la realizabilidad de la trayectoria (no choca con ningún obstáculo)

  - Clase MySegment:
    - Punto inicial y final (o punto inicial y vector)
    - Función bool Crosses(const MySegment& other); para comprobar el cruce de 2 segmentos

    (https://stackoverflow.com/questions/4977491/determining-if-two-line-segments-intersect/4977569#4977569)
  - Un segmento de la trayectoria choca con un obstáculo si se cruza con alguno de los segmentos que forman sus bordes