libavoid

Generated by Doxygen 1.9.1

Chapter 1

libavoid: Object-avoiding orthogonal and polyline connector routing library

libavoid is a cross-platform C++ library providing fast, object-avoiding orthogonal and polyline connector routing for use in interactive diagram editors.

libavoid is part of the `Adaptagrams project`. There are no official releases yet, though the code is stable and available from the Adaptagrams `GitHub repository`.

The API is documented using Doxygen. The documentation you are currently reading can be obtained by running doxygen in the cola or libavoid directory. There is also a simple `documented example` to help you get started.

libavoid is written and maintained by `Michael Wybrow`, a member of `Immersive Analytics Lab` at Monash University, Australia.

The algorithms used for the connector routing are described in the following papers. If you use libavoid, please cite the relevant paper.

- M. Wybrow, K. Marriott, and P.J. Stuckey. Orthogonal connector routing.
  In Proceedings of 17th International Symposium on Graph Drawing (GD '09),
  LNCS 5849, pages 219–231. Spring-Verlag, 2010. [`DOI`] [`PDF`]
- M. Wybrow, K. Marriott, and P.J. Stuckey. Incremental connector routing.
  In Proceedings of 13th International Symposium on Graph Drawing (GD '05),
  LNCS 3843, pages 446—457. Springer-Verlag, 2006. [`DOI`] [`PDF`]
- M. Wybrow, K. Marriott and P.J. Stuckey. Orthogonal hyperedge routing.
  In Proceedings of 7th International Conference on the Theory and Application of Diagrams (Diagrams 2012),
  LNCS (LNAI) 7352, pages 51–64. Springer-Verlag, 2012. [`DOI`] [`PDF`]
- K. Marriott, P.J. Stuckey, and M. Wybrow. Seeing Around Corners: Fast Orthogonal Connector Routing.
  In Proceedings of the 8th International Conference on the Theory and Application of Diagrams (Diagrams 2014),
  LNCS 8578, pages 31–37. Springer-Verlag, 2014. [`DOI`] [`PDF`]

libavoid is currently used in the following software:

- A commercial circuit diagram editor;
- `Dunnart`, a prototype research constraint-based diagram editor;

- Inkscape, the popular Open Source vector graphics editor;
- Arcadia, a visualisation tool for metabolic pathways;
- Gaphas, an open source diagramming widget for GTK+, written in Python;
- BRL-CAD, Constructive Solid Geometry (CSG) solid modeling system;
- QxOrm, a C++ library designed to provide Object Relational Mapping (ORM) feature to C++ users; and
- QxEntityEditor, a graphic editor for QxOrm library, providing a graphic way to manage a data model.

Chapter 2

libavoid — Documented code example

libavoid is a C++ library. Its code is all within the Avoid namespace.

First, you must create an instance of the router.

```
Avoid::Router *router = new Avoid::Router(Avoid::PolyLineRouting);
```

To add a shape (obstacle) to the router, you first create a Avoid::ShapeRef by giving the bounding box of the obstacle. This adds the shape to the router (and cause rerouting of connectors it intersects). It also passes ownership of the shapeRef object to the router instance, though it is still fine for you to keep a reference to it.

```
// Create the ShapeRef:
Avoid::Rectangle rectangle(Avoid::Point(20.0, 35.0), Avoid::Point(40.0, 12.0));
Avoid::ShapeRef *shapeRef = new Avoid::ShapeRef(router, rectangle);
```

or

```
Avoid::Polygon shapePoly(3);
shapePoly.ps[0] = Avoid::Point(1.0, 1.0);
shapePoly.ps[1] = Avoid::Point(2.5, 1.5);
shapePoly.ps[2] = Avoid::Point(1.5, 2.5);
Avoid::ShapeRef *shapeRef = new Avoid::ShapeRef(router, shapePoly);
```

The relevant prototypes (all in the Avoid namespace) are as follows. If a shape ID is specified, it should be non-zero and unique among all shapes and connectors.

```
Avoid::Rectangle(const Avoid::Point& topLeft, const Avoid::Point& bottomRight);
Avoid::Rectangle(const Avoid::Point& centre, const double width, const double height);
Avoid::ShapeRef(Avoid::Router *router, const Avoid::Polygon& polygon, unsigned int id = 0);
```

To move or resize a shape already in the router, you do the following:

```
router->moveShape(shapeRef, newPolygon);
```

or

```
double xmove = 20, ymove = 15;
router->moveShape(shapeRef, xmove, ymove);
```

In its default mode the router will queue multiple shape movements and perform the changes to the visibility graph in an optimised order. Thus you make several calls to Avoid::Router::moveShape() for different shapes and then tell the router to process the moves. This tend to be useful in interactive applications where the user may move multiple shapes at once.

```
router->moveShape(shapeRef1, newPolygon1);
router->moveShape(shapeRef2, newPolygon2);
router->processTransaction();
```

To delete a shape from the router (and reroute connectors that then have a better path) you do the following.

```
router->deleteShape(shapeRef);
```

This will cause the router to free the memory for the shapeRef. You should discard your reference to the shapeRef after this call.

To add a new connector to the router, you do the following:
```
Avoid::ConnEnd srcPt(Avoid::Point(1.2, 0.5));
Avoid::ConnEnd dstPt(Avoid::Point(3.0, 4.0));
Avoid::ConnRef *connRef = new Avoid::ConnRef(router, srcPt, dstPt);
```

This passes ownership of the connRef object to the router instance, though it is still fine for you to keep a reference to it.

To remove a connector from the router:
```
router->deleteConnector(connRef);
```

This will cause the router to free the memory for the connRef. You should discard your reference to the connRef after this call.

You can set a function to be called when the connector needs to be redrawn. When called, this function will be passed the pointer given as a second argument to Avoid::ConnRef::setCallback():
```
void connCallback(void *ptr)
{
    Avoid::ConnRef *connRef = (Avoid::ConnRef *) ptr;
    printf("Connector %u needs rerouting!\n", connRef->id());
}
connRef->setCallback(connCallback, connRef);
```

The callback will be triggered by movement, addition and deletion of shapes, as well as by adjustment of the connector endpoints, or by processing a transaction that includes any of these events. You can check if a connector path has changed, and hence the object requires repainting (say because a better path is available due to a shape being deleted):
```
if (connRef->needsRepaint()) ...
```

If you want to trigger the callback for a connector after moving its endpoints (or when it is first created you can do this via:
```
connRef->processTransaction();
```

You can then get the new path as follows:
```
const Avoid::PolyLine route = connRef->dispayRoute();
for (size_t i = 0; i &lt; route.size(); ++i)
{
    Avoid::Point point = route.at(i);
    printf("%f, %f\n", point.x, point.y);
}
```

Obviously the alternative to using the callback mechanism is to iterate through all connectors and check their needsRepaint() value after having called processTransaction().

You can update the endpoints of a connector with:
```
Avoid::ConnEnd newSrcPt(Avoid::Point(6, 3));
Avoid::ConnEnd newDstPt(Avoid::Point(12, 67));
connRef->setEndpoints(newSrcPt, newDstPt);
```

or
```
Avoid::ConnEnd newSrcPt(Avoid::Point(6, 3));
connRef->setSourceEndpoint(newSrcPt);
Avoid::ConnEnd newDstPt(Avoid::Point(6, 3));
connRef->setDestEndpoint(newDstPt);
```

You can also create connection pins on shapes and attach connectors directly to these. Then when you move or resize the shapes, the connector endpoints attached to them will be automatically rerouted.

You can create a connection pin as follows:
```
const unsigned int CENTRE = 1;
new Avoid::ShapeConnectionPin(shapeRef, CENTRE, Avoid::ATTACH_POS_CENTRE, Avoid::ATTACH_POS_CENTRE);
```

This one connects to the centre of the shape, but the position can be specified anywhere within the shape as a proportion of the shape's width and height.

You can then attach a connector to the connection pin be doing the following:
```
Avoid::ConnEnd newSrcPt(shapeRef, CENTRE);
connRef->setSourceEndpoint(newSrcPt);
```

See also a short example: example.cpp in the libavoid/tests directory

Chapter 3

Namespace Index

3.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 4

Hierarchical Index

4.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 5

Class Index

5.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 6

File Index

6.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 7

Namespace Documentation

7.1 Avoid Namespace Reference

libavoid: Object-avoiding orthogonal and polyline connector routing library.

Classes

- class ShapeConnectionPin

  The ShapeConnectionPin class represents a fixed point or "pin" on a shape that can be connected to.
- class Checkpoint

  A checkpoint is a point that the route for a particular connector must visit. They may optionally be given an arrival/departure direction.
- class ConnRef

  The ConnRef class represents a connector object.
- class ConnEnd

  The ConnEnd class represents different possible endpoints for connectors.
- class Point

  The Point class defines a point in the plane.
- class Box

  A bounding box, represented by the top-left and bottom-right corners.
- class PolygonInterface

  A common interface used by the Polygon classes.
- class Edge

  A line between two points.
- class Polygon

  A dynamic Polygon, to which points can be easily added and removed.
- class ReferencingPolygon

  A Polygon which just references its points from other Polygons.
- class Rectangle

  A Rectangle, a simpler way to define the polygon for square or rectangular shapes.
- struct HyperedgeNewAndDeletedObjectLists

  The HyperedgeNewAndDeletedObjectLists class stores lists of objects created and deleted during hyperedge improvement.
- class HyperedgeRerouter

The HyperedgeRerouter class is a convenience object that can be used to register hyperedges to be rerouted, improving the placement of their junctions and connector paths.

- class JunctionRef

  The JunctionRef class represents a fixed or free-floating point that connectors can be attached to.

- class Router

  The Router class represents a libavoid router instance.
- class ShapeRef

  The ShapeRef class represents a shape object.
- class ClusterRef

  The ClusterRef class represents a cluster object.

Typedefs

- typedef std::list< ConnRef ∗ > ConnRefList

  A list of ConnRef objects.
- typedef unsigned int ConnDirFlags

  One or more Avoid::ConnDirFlag options.
- typedef Point Vector

  A vector, represented by the Point class.
- typedef Polygon PolyLine

  A multi-segment line, represented with the Polygon class.
- typedef std::list< ConnEnd > ConnEndList

  A list of ConnEnd objects.
- typedef std::list< JunctionRef ∗ > JunctionRefList

  A list of JunctionRef objects.

Enumerations

- enum ConnType { }

  Describes the type of routing that is performed for each connector.
- enum ConnDirFlag { }

  Flags that can be passed to the ConnEnd constructor to specify which sides of a shape this point should have visibility to if it is located within the shape's area.
- enum ConnEndType { ConnEndPoint , ConnEndShapePin , ConnEndJunction , ConnEndEmpty }

  Types that describe the kind a connection that a ConnEnd represents.
- enum RouterFlag { PolyLineRouting = 1 , OrthogonalRouting = 2 }

  Flags that can be passed to the router during initialisation to specify options.
- enum RoutingParameter {
  segmentPenalty = 0 , anglePenalty , crossingPenalty , clusterCrossingPenalty ,
  fixedSharedPathPenalty , portDirectionPenalty , shapeBufferDistance , idealNudgingDistance ,
  reverseDirectionPenalty , **lastRoutingParameterMarker** }

  Types of routing parameters and penalties that can be used to tailor the style and improve the quality of the connector routes produced.
- enum RoutingOption {
  nudgeOrthogonalSegmentsConnectedToShapes = 0 , improveHyperedgeRoutesMovingJunctions ,
  penaliseOrthogonalSharedPathsAtConnEnds , nudgeOrthogonalTouchingColinearSegments ,
  performUnifyingNudgingPreprocessingStep , improveHyperedgeRoutesMovingAddingAndDeletingJunctions
  , nudgeSharedPathsWithCommonEndPoint , **lastRoutingOptionMarker** }

  Types of routing options that can be enabled.

- enum TransactionPhases {
TransactionPhaseOrthogonalVisibilityGraphScanX = 1 , TransactionPhaseOrthogonalVisibilityGraphScanY ,
TransactionPhaseRouteSearch , TransactionPhaseCrossingDetection ,
TransactionPhaseRerouteSearch , TransactionPhaseOrthogonalNudgingX , TransactionPhaseOrthogonalNudgingY
, TransactionPhaseCompleted }

  Types of routing phases reported by Router::shouldContinueTransactionWithProgress().
- enum ShapeTransformationType {
TransformationType_CW90 = 0 , TransformationType_CW180 = 1 , TransformationType_CW270 = 2 ,
TransformationType_FlipX = 3 ,
TransformationType_FlipY = 4 }

  Describes the type of transformation that has been applied to a shape having its transformConnectionPinPositions() method called.

7.1.1 Detailed Description

libavoid: Object-avoiding orthogonal and polyline connector routing library.

You should use libavoid via an instance of the Router class.

7.1.2 Enumeration Type Documentation

7.1.2.1 ConnDirFlag

```
enum Avoid::ConnDirFlag
```

Flags that can be passed to the ConnEnd constructor to specify which sides of a shape this point should have visibility to if it is located within the shape's area.

Like SVG, libavoid considers the Y-axis to point downwards, that is, like screen coordinates the coordinates increase from left-to-right and also from top-to-bottom.

Enumerator

ConnDirUp	This option specifies the point should be given visibility to the top of the shape that it is located within.
ConnDirDown	This option specifies the point should be given visibility to the bottom of the shape that it is located within.
ConnDirLeft	This option specifies the point should be given visibility to the left side of the shape that it is located within.
ConnDirRight	This option specifies the point should be given visibility to the right side of the shape that it is located within.
ConnDirAll	This option, provided for convenience, specifies the point should be given visibility to all four sides of the shape that it is located within.

7.1.2.2 ConnEndType

```
enum Avoid::ConnEndType
```

Types that describe the kind a connection that a ConnEnd represents.

Enumerator

ConnEndPoint	The ConnEnd represents a free-floating point that may or may not have visibility in specific directions.
ConnEndShapePin	The ConnEnd attaches to a specific ShapeConnectionPin on a shape.
ConnEndJunction	The ConnEnd attaches to a junction.
ConnEndEmpty	The ConnEnd is empty and doesn't have any information set.

7.1.2.3 ConnType

```
enum Avoid::ConnType
```

Describes the type of routing that is performed for each connector.

Enumerator

ConnType_PolyLine	The connector path will be a shortest-path poly-line that routes around obstacles.
ConnType_Orthogonal	The connector path will be a shortest-path orthogonal poly-line (only vertical and horizontal line segments) that routes around obstacles.

7.1.2.4 RouterFlag

```
enum Avoid::RouterFlag
```

Flags that can be passed to the router during initialisation to specify options.

Enumerator

PolyLineRouting	This option specifies that the router should maintain the structures necessary to allow poly-line connector routing.
OrthogonalRouting	This option specifies that the router should maintain the structures necessary to allow orthogonal connector routing.

7.1.2.5 RoutingOption

```
enum Avoid::RoutingOption
```

Types of routing options that can be enabled.

Enumerator

nudgeOrthogonalSegmentsConnectedToShapes	This option causes the final segments of connectors, which are attached to shapes, to be nudged apart. Usually these segments are fixed, since they are considered to be attached to ports. Defaults to false. This option also causes routes running through the same checkpoint to be nudged apart. This option has no effect if nudgeSharedPathsWithCommonEndPoint is set to false, **Note** This will allow routes to be nudged up to the bounds of shapes.
improveHyperedgeRoutesMovingJunctions	This option causes hyperedge routes to be locally improved fixing obviously bad paths. As part of this process libavoid will effectively move junctions, setting new ideal positions which can be accessed via JunctionRef::recommendedPosition() for each junction. Defaults to true. This will not add or remove junctions, so will keep the hyperedge topology the same. Better routes can be achieved by enabling the improveHyperedgeRoutesMovingAddingAndDeletingJunctions option. If initial sensible positions for junctions in hyperedges are not known you can register those hyperedges with the HyperedgeRerouter class for complete rerouting. **See also** improveHyperedgeRoutesMovingAddingAndDeletingJunctions Router::hyperedgeRerouter()
penaliseOrthogonalSharedPathsAtConnEnds	This option penalises and attempts to reroute orthogonal shared connector paths terminating at a common junction or shape connection pin. When multiple connector paths enter or leave the same side of a junction (or shape pin), the router will attempt to reroute these to different sides of the junction or different shape pins. Defaults to false. This option depends on the fixedSharedPathPenalty penalty having been set. **See also** fixedSharedPathPenalty **Note** This option is still experimental! It is not recommended for normal use.

Enumerator

nudgeOrthogonalTouchingColinearSegments	This option can be used to control whether collinear line segments that touch just at their ends will be nudged apart. The overlap will usually be resolved in the other dimension, so this is not usually required. Defaults to false.
performUnifyingNudgingPreprocessingStep	This option can be used to control whether the router performs a preprocessing step before orthogonal nudging where is tries to unify segments and centre them in free space. This generally results in better quality ordering and nudging. Defaults to true. You may wish to turn this off for large examples where it can be very slow and will make little difference.
improveHyperedgeRoutesMovingAddingAnd↩ DeletingJunctions	This option causes hyperedge routes to be locally improved fixing obviously bad paths. It can cause junctions and connectors to be added or removed from hyperedges. To get details of these changes for each connector you can call Router::newAndDeletedObjectListsFromHyperedgeImprovement(). As part of this process libavoid will effectively move junctions by setting new ideal positions for each remaining or added junction, which can be read from JunctionRef::recommendedPosition() for each junction. Defaults to false. If set, this option overrides the improveHyperedgeRoutesMovingJunctions option. If initial sensible positions for junctions in hyperedges are not known you can register those hyperedges with the HyperedgeRerouter class for complete rerouting. **See also** improveHyperedgeRoutesMovingJunctions Router::hyperedgeRerouter()
nudgeSharedPathsWithCommonEndPoint	This option determines whether intermediate segments of connectors that are attached to common endpoints will be nudged apart. Usually these segments get nudged apart, but you may want to turn this off if you would prefer that entire shared paths terminating at a common end point should overlap. Defaults to true.

7.1.2.6 RoutingParameter

```
enum Avoid::RoutingParameter
```

Types of routing parameters and penalties that can be used to tailor the style and improve the quality of the connector routes produced.

Enumerator

segmentPenalty	This penalty is applied for each segment in the connector path beyond the first. This should always normally be set when doing orthogonal routing to prevent step-like connector paths. **Note** This penalty must be set (i.e., be greater than zero) in order for orthogonal connector nudging to be performed, since this requires reasonable initial routes.
anglePenalty	This penalty is applied in its full amount to tight acute bends in the connector path. A smaller portion of the penalty is applied for slight bends, i.e., where the bend is close to 180 degrees. This is useful for polyline routing where there is some evidence that tighter corners are worse for readability, but that slight bends might not be so bad, especially when smoothed by curves.
crossingPenalty	This penalty is applied whenever a connector path crosses another connector path. It takes shared paths into consideration and the penalty is only applied if there is an actual crossing. **Note** This penalty is still experimental! It is not recommended for normal use.
clusterCrossingPenalty	This penalty is applied whenever a connector path crosses a cluster boundary. **Note** This penalty is still experimental! It is not recommended for normal use. This penalty is very slow. You can override the method Router::shouldContinueTransactionWithProgress() to check progress and possibly cancel overly slow transactions.
fixedSharedPathPenalty	This penalty is applied whenever a connector path shares some segments with an immovable portion of an existing connector route (such as the first or last segment of a connector). **Note** This penalty is still experimental! It is not recommended for normal use.
portDirectionPenalty	This penalty is applied to port selection choice when the other end of the connector being routed does not appear in any of the 90 degree visibility cones centered on the visibility directions for the port. **Note** This penalty is still experimental! It is not recommended for normal use. This penalty is very slow. You can override the method Router::shouldContinueTransactionWithProgress() to check progress and possibly cancel overly slow transactions.
shapeBufferDistance	This parameter defines the spacing distance that will be added to the sides of each shape when determining obstacle sizes for routing. This controls how closely connectors pass shapes, and can be used to prevent connectors overlapping with shape boundaries. By default, this distance is set to a value of 0.
idealNudgingDistance	This parameter defines the spacing distance that will be used for nudging apart overlapping corners and line segments of connectors. By default, this distance is set to a value of 4.

Enumerator

reverseDirectionPenalty	This penalty is applied whenever a connector path travels in the direction opposite of the destination from the source endpoint. By default this penalty is set to zero. This shouldn't be needed in most cases but can be useful if you use penalties such as crossingPenalty which cause connectors to loop around obstacles.

7.1.2.7 ShapeTransformationType

```
enum Avoid::ShapeTransformationType
```

Describes the type of transformation that has been applied to a shape having its transformConnectionPinPositions() method called.

Enumerator

TransformationType_CW90	The shape has been rotated clockwise by 90 degrees.
TransformationType_CW180	The shape has been rotated clockwise by 180 degrees.
TransformationType_CW270	The shape has been rotated clockwise by 270 degrees.
TransformationType_FlipX	The shape has been flipped horizontally in the X-dimension.
TransformationType_FlipY	The shape has been flipped vertically in the Y-dimension.

7.1.2.8 TransactionPhases

```
enum Avoid::TransactionPhases
```

Types of routing phases reported by Router::shouldContinueTransactionWithProgress().

This phases will occur in the order given here, but each phase may take varying amounts of time.

Enumerator

TransactionPhaseOrthogonalVisibilityGraphScanX	The orthogonal visibility graph is built by conducting a scan in each dimension. This is the x-dimension.
TransactionPhaseOrthogonalVisibilityGraphScanY	The orthogonal visibility graph is built by conducting a scan in each dimension. This is the y-dimension.
TransactionPhaseRouteSearch	Initial routes are searched for in the visibility graph.
TransactionPhaseCrossingDetection	With crossing penalties enabled, crossing detection is performed to find all crossings.
TransactionPhaseRerouteSearch	Crossing connectors are rerouted to search for better routes.
TransactionPhaseOrthogonalNudgingX	Orthogonal edge segments are nudged apart in the x-dimension.
TransactionPhaseOrthogonalNudgingY	Orthogonal edge segments are nudged apart in the y-dimension.
TransactionPhaseCompleted	Not a real phase, but represents the router is finished (or has aborted) the transaction and you may interact with is again.

Chapter 8

Class Documentation

8.1 Avoid::Box Class Reference

A bounding box, represented by the top-left and bottom-right corners.

```
#include <geomtypes.h>
```

Public Attributes

- Point min

  The top-left point.
- Point max

  The bottom-right point.

8.1.1 Detailed Description

A bounding box, represented by the top-left and bottom-right corners.

The documentation for this class was generated from the following files:

- geomtypes.h
- geomtypes.cpp

8.2 Avoid::Checkpoint Class Reference

A checkpoint is a point that the route for a particular connector must visit. They may optionally be given an arrival/departure direction.

```
#include <connector.h>
```

Public Member Functions

- Checkpoint (const Point &p)

    A point that a route must visit.
- Checkpoint (const Point &p, ConnDirFlags ad, ConnDirFlags dd)

    A point that a route must visit.

8.2.1 Detailed Description

A checkpoint is a point that the route for a particular connector must visit. They may optionally be given an arrival/departure direction.

8.2.2 Constructor & Destructor Documentation

8.2.2.1 Checkpoint() [1/2]

```
Avoid::Checkpoint::Checkpoint (
             const Point & p )  [inline]
```

A point that a route must visit.

The connector will be able to enter and leave this checkpoint from any direction.

Parameters

in	*p*	The Point that must be visited.

8.2.2.2 Checkpoint() [2/2]

```
Avoid::Checkpoint::Checkpoint (
             const Point & p,
             ConnDirFlags ad,
             ConnDirFlags dd )  [inline]
```

A point that a route must visit.

The connector will be able to enter and leave this checkpoint from the directions specified. Give Avoid::ConnDirAll to specify all directions.

Parameters

in	*p*	The Point that must be visited.
in	*ad*	Avoid::ConnDirFlags denoting arrival directions for the connector portion leading up to this checkpoint.
in	*dd*	Avoid::ConnDirFlags denoting departure directions for the connector portion leading away from this checkpoint.

The documentation for this class was generated from the following file:

- connector.h

8.3 Avoid::ClusterRef Class Reference

The ClusterRef class represents a cluster object.

```
#include <viscluster.h>
```

Public Member Functions

- ClusterRef (Router ∗router, Polygon &poly, const unsigned int id=0)

  Cluster reference constructor.
- ∼ClusterRef ()

  Cluster reference destructor.
- void setNewPoly (Polygon &poly)

  Update the polygon boundary for this cluster.
- unsigned int id (void) const

  Returns the ID of this cluster.
- ReferencingPolygon & polygon (void)

  Returns a reference to the polygon boundary of this cluster.
- Polygon & rectangularPolygon (void)

  Returns a reference to the rectangular boundary of this cluster.
- Router ∗ router (void) const

  Returns a pointer to the router scene this cluster is in.

8.3.1 Detailed Description

The ClusterRef class represents a cluster object.

Cluster are boundaries around groups of shape objects. Ideally, only connectors with one endpoint inside the cluster and one endpoint outside the cluster will cross the cluster boundary. Connectors that begin and end inside a cluster will not route outside it, and connectors that begin and end outside the cluster will not enter the cluster.

Note

While the functionality of this class works, it is currently experimental you will likely suffer a large performance hit when using it.

8.3.2 Constructor & Destructor Documentation

8.3.2.1 ClusterRef()

```
Avoid::ClusterRef::ClusterRef (
             Router * router,
             Polygon & poly,
             const unsigned int id = 0 )
```

Cluster reference constructor.

Creates a cluster object reference, but does not yet place it into the Router scene. You can add or remove the cluster to/from the scene with Router::addCluster() and Router::delCluster(). The cluster can effectively be moved with ClusterRef::setNewPoly() method.

The poly argument should be used to specify a polygon boundary. The rectangular boundary will be automatically generated from this. The polygon boundary could be a convex hull consisting of points from the boundaries of shapes.

Note

Regarding IDs: You can let libavoid manually handle IDs by not specifying them. Alternatively, you can specify all IDs yourself, but you must be careful to makes sure that each object in the scene (shape, connector, cluster, etc) is given a unique, positive ID. This uniqueness is checked if assertions are enabled, but if not and there are clashes then strange things can happen.

Parameters

in	*router*	The router scene to place the cluster into.
in	*poly*	A Polygon representing the boundary of the cluster.
in	*id*	Optionally, a positive integer ID unique among all objects.

8.3.3 Member Function Documentation

8.3.3.1 id()

```
unsigned int Avoid::ClusterRef::id (
             void ) const
```

Returns the ID of this cluster.

Returns

The ID of the cluster.

8.3.3.2 polygon()

```
ReferencingPolygon & Avoid::ClusterRef::polygon (
            void  )
```

Returns a reference to the polygon boundary of this cluster.

Returns

A reference to the polygon boundary of the cluster.

8.3.3.3 rectangularPolygon()

```
Polygon & Avoid::ClusterRef::rectangularPolygon (
            void  )
```

Returns a reference to the rectangular boundary of this cluster.

Returns

A reference to the rectangular boundary of the cluster.

8.3.3.4 router()

```
Router * Avoid::ClusterRef::router (
            void  ) const
```

Returns a pointer to the router scene this cluster is in.

Returns

A pointer to the router scene for this cluster.

8.3.3.5 setNewPoly()

```
void Avoid::ClusterRef::setNewPoly (
            Polygon & poly )
```

Update the polygon boundary for this cluster.

You should specify a polygon boundary. The rectangular one will be generated automatically from this.

Parameters

`in`	*poly*	A Polygon representing the boundary of the cluster.

The documentation for this class was generated from the following files:

- viscluster.h
- viscluster.cpp

8.4 Avoid::ConnEnd Class Reference

The ConnEnd class represents different possible endpoints for connectors.

```
#include <connend.h>
```

Public Member Functions

- ConnEnd (const Point &point)

  Constructs a ConnEnd from a free-floating point.
- ConnEnd (const Point &point, const ConnDirFlags visDirs)

  Constructs a ConnEnd from a free-floating point as well as a set of flags specifying visibility for this point if it is located inside a shape.
- ConnEnd (ShapeRef ∗shapeRef, const unsigned int connectionPinClassID)

  Constructs a ConnEnd attached to one of a particular set of connection pins on a shape.
- ConnEnd (JunctionRef ∗junctionRef)

  Constructs a ConnEnd attached to one of the connection pins on a junction.
- ConnEndType type (void) const

  Returns the kind of connection this ConnEnd represents.
- const Point position (void) const

  Returns the position of this connector endpoint.
- ConnDirFlags directions (void) const

  Returns the directions in which this connector endpoint should be given visibility.
- ShapeRef ∗ shape (void) const

  Returns the shape this ConnEnd attaches to, or nullptr.
- JunctionRef ∗ junction (void) const

  Returns the junction this ConnEnd attaches to, or nullptr.
- unsigned int pinClassId (void) const

  Returns the pin class ID for a ConnEnd attached to a shape.

8.4.1 Detailed Description

The ConnEnd class represents different possible endpoints for connectors.

ConnEnds may be free-floating points, points attached to junctions (between multiple connectors), or points attached to shapes (either to the centre of the shape or to particular pin positions on the shape).

8.4.2 Constructor & Destructor Documentation

8.4.2.1 ConnEnd() [1/4]

```
Avoid::ConnEnd::ConnEnd (
            const Point & point )
```

Constructs a ConnEnd from a free-floating point.

Parameters

in	*point*	The position of the connector endpoint.

8.4.2.2 ConnEnd() [2/4]

```
Avoid::ConnEnd::ConnEnd (
            const Point & point,
            const ConnDirFlags visDirs )
```

Constructs a ConnEnd from a free-floating point as well as a set of flags specifying visibility for this point if it is located inside a shape.

Parameters

in	*point*	The position of the connector endpoint.
in	*visDirs*	One or more Avoid::ConnDirFlag options specifying the directions that this point should be given visibility if it is inside a shape. Currently has no effect if outside of shapes.

8.4.2.3 ConnEnd() [3/4]

```
Avoid::ConnEnd::ConnEnd (
            ShapeRef * shapeRef,
            const unsigned int connectionPinClassID )
```

Constructs a ConnEnd attached to one of a particular set of connection pins on a shape.

This is the recommended method for connecting to shapes that may later be moved or resized and for which you don't want to track and specify the connector endpoints yourself. See the ShapeConnectionPin documentation for more information.

If you to just connect to the centre of a shape you should just create a centre connection pin
```
const unsigned int CENTRE = 1;
new Avoid::ShapeConnectionPin(shapeRef, CENTRE, Avoid::ATTACH_POS_CENTRE, Avoid::ATTACH_POS_CENTRE);
```

and attach to that with

```
ConnEnd(shapeRef, CENTRE);
```

If a pin with the specified pin class ID doesn't exist then you will get a warning and a straight-line path between the source and destination endpoints of the connector will be returned during routing.

Parameters

in	*shapeRef*	A pointer to the containing shape's ShapeRef.
in	*connectionPinClassID*	A non-zero integer denoting the class ID for the set of pins to connect to.

8.4.2.4 ConnEnd() [4/4]

```
Avoid::ConnEnd::ConnEnd (
             JunctionRef * junctionRef )
```

Constructs a ConnEnd attached to one of the connection pins on a junction.

This is the recommended method for connecting to junctions that may later be moved. See the ShapeConnectionPin documentation for more information.

Parameters

in	*junctionRef*	A pointer to the containing junction's junctionRef.

8.4.3 Member Function Documentation

8.4.3.1 directions()

```
ConnDirFlags Avoid::ConnEnd::directions (
             void  ) const
```

Returns the directions in which this connector endpoint should be given visibility.

Returns

The visibility directions for this connector endpoint.

8.4.3.2 junction()

```
JunctionRef * Avoid::ConnEnd::junction (
             void  ) const
```

Returns the junction this ConnEnd attaches to, or nullptr.

Will be valid only if type() == ConnEndJunction.

Returns

The JunctionRef pointer that the ConnEnd attaches to, or nullptr.

8.4.3.3 pinClassId()

```
unsigned int Avoid::ConnEnd::pinClassId (
            void ) const
```

Returns the pin class ID for a ConnEnd attached to a shape.

Will be valid only if type() == ConnEndShapePin.

Returns

An unsigned int representing the pin class ID for the ConnEnd.

8.4.3.4 position()

```
const Point Avoid::ConnEnd::position (
            void ) const
```

Returns the position of this connector endpoint.

Returns

The position of this connector endpoint.

8.4.3.5 shape()

```
ShapeRef * Avoid::ConnEnd::shape (
            void ) const
```

Returns the shape this ConnEnd attaches to, or nullptr.

Will be valid only if type() == ConnEndShapePin.

Returns

The ShapeRef pointer that the ConnEnd attaches to, or nullptr.

8.4.3.6 type()

```
ConnEndType Avoid::ConnEnd::type (
            void ) const
```

Returns the kind of connection this ConnEnd represents.

Returns

The ConnEndType represented by this ConnEnd.

The documentation for this class was generated from the following files:

- connend.h
- connend.cpp

8.5 Avoid::ConnRef Class Reference

The ConnRef class represents a connector object.

```
#include <connector.h>
```

Public Member Functions

- ConnRef (Router ∗router, const unsigned int id=0)

    Constructs a connector with no endpoints specified.
- ConnRef (Router ∗router, const ConnEnd &src, const ConnEnd &dst, const unsigned int id=0)

    Constructs a connector with endpoints specified.
- ∼ConnRef ()

    Connector reference destuctor.
- void setEndpoints (const ConnEnd &srcPoint, const ConnEnd &dstPoint)

    Sets both a new source and destination endpoint for this connector.
- void setSourceEndpoint (const ConnEnd &srcPoint)

    Sets just a new source endpoint for this connector.
- void setDestEndpoint (const ConnEnd &dstPoint)

    Sets just a new destination endpoint for this connector.
- unsigned int id (void) const

    Returns the ID of this connector.
- Router ∗ router (void) const

    Returns a pointer to the router scene this connector is in.
- bool needsRepaint (void) const

    Returns an indication of whether this connector has a new route and thus needs to be repainted.
- const PolyLine & route (void) const

    Returns a reference to the current raw "debug" route for the connector.
- PolyLine & displayRoute (void)

    Returns a reference to the current display version of the route for the connector.
- void setCallback (void(∗cb)(void ∗), void ∗ptr)

    Sets a callback function that will called to indicate that the connector needs rerouting.
- ConnType routingType (void) const

    Returns the type of routing performed for this connector.
- void setRoutingType (ConnType type)

    Sets the type of routing to be performed for this connector.
- std::pair< JunctionRef ∗, ConnRef ∗ > splitAtSegment (const size_t segmentN)

    Splits a connector in the centre of the segmentNth segment and creates a junction point there as well as a second connector.
- void setRoutingCheckpoints (const std::vector< Checkpoint > &checkpoints)

    Allows the user to specify a set of checkpoints that this connector will route via.
- std::vector< Checkpoint > routingCheckpoints (void) const

    Returns the current set of routing checkpoints for this connector.
- std::pair< ConnEnd, ConnEnd > endpointConnEnds (void) const

    Returns ConnEnds specifying what this connector is attached to.
- void setFixedRoute (const PolyLine &route)

    Sets a fixed user-specified route for this connector.
- void setFixedExistingRoute (void)

    Sets a fixed existing route for this connector.
- bool hasFixedRoute (void) const

    Returns whether the connector route is marked as fixed.
- void clearFixedRoute (void)

    Returns the connector to being automatically routed if it was marked as fixed.

8.5.1 Detailed Description

The ConnRef class represents a connector object.

Connectors are a (possible multi-segment) line between two points. They are routed intelligently so as not to overlap any of the shape objects in the Router scene.

Routing penalties can be applied, resulting in more aesthetically pleasing connector paths with fewer segments or less severe bend-points.

You can set a function to be called when the connector has been rerouted and needs to be redrawn. Alternatively, you can query the connector's needsRepaint() function to determine this manually.

Usually, it is expected that you would create a ConnRef for each connector in your diagram and keep that reference in your own connector class.

8.5.2 Constructor & Destructor Documentation

8.5.2.1 ConnRef() `[1/2]`

```
Avoid::ConnRef::ConnRef (
            Router * router,
            const unsigned int id = 0 )
```

Constructs a connector with no endpoints specified.

The constructor requires a valid Router instance. This router will take ownership of the connector. Hence, you should not call the destructor yourself, but should instead call Router::deleteConnector() and the router instance will remove and then free the connector's memory.

Note

Regarding IDs: You can let libavoid manually handle IDs by not specifying them. Alternatively, you can specify all IDs yourself, but you must be careful to makes sure that each object in the scene (shape, connector, cluster, etc) is given a unique, positive ID. This uniqueness is checked if assertions are enabled, but if not and there are clashes then strange things can happen.

Parameters

in	*router*	The router scene to place the connector into.
in	*id*	Optionally, a positive integer ID unique among all objects.

8.5.2.2 ConnRef() `[2/2]`

```
Avoid::ConnRef::ConnRef (
            Router * router,
            const ConnEnd & src,
            const ConnEnd & dst,
            const unsigned int id = 0 )
```

Constructs a connector with endpoints specified.

The constructor requires a valid Router instance. This router will take ownership of the connector. Hence, you should not call the destructor yourself, but should instead call Router::deleteConnector() and the router instance will remove and then free the connector's memory.

If an ID is not specified, then one will be assigned to the shape. If assigning an ID yourself, note that it should be a unique positive integer. Also, IDs are given to all objects in a scene, so the same ID cannot be given to a shape and a connector for example.

Parameters

in	*router*	The router scene to place the connector into.
in	*id*	A unique positive integer ID for the connector.
in	*src*	The source endpoint of the connector.
in	*dst*	The destination endpoint of the connector.

8.5.2.3 ∼ConnRef()

```
Avoid::ConnRef::∼ConnRef ( )
```

Connector reference destuctor.

Do not call this yourself, instead call Router::deleteConnector(). Ownership of this object belongs to the router scene.

8.5.3 Member Function Documentation

8.5.3.1 clearFixedRoute()

```
void Avoid::ConnRef::clearFixedRoute (
            void )
```

Returns the connector to being automatically routed if it was marked as fixed.

See also

setFixedRoute()

8.5.3.2 displayRoute()

```
Polygon & Avoid::ConnRef::displayRoute (
             void  )
```

Returns a reference to the current display version of the route for the connector.

The display version of a route has been simplified to collapse all collinear line segments into single segments. It also has all post-processing applied to the route, including centering, curved corners and nudging apart of overlapping segments.

Returns

The PolyLine display route for the connector.

8.5.3.3 endpointConnEnds()

```
std::pair< ConnEnd, ConnEnd > Avoid::ConnRef::endpointConnEnds (
             void  ) const
```

Returns ConnEnds specifying what this connector is attached to.

This may be useful during hyperedge rerouting. You can check the type and properties of the ConnEnd objects to find out what this connector is attached to. The ConnEnd::type() will be ConnEndEmpty if the connector has not had its endpoints initialised.

Note

If the router is using transactions, you might get unexpected results if you call this after changing objects but before calling Router::processTransaction(). In this case changes to ConnEnds for the connector may be queued and not yet applied, so you will get old (or empty) values.

Returns

A pair of ConnEnd objects specifying what the connector is attached to.

8.5.3.4 hasFixedRoute()

```
bool Avoid::ConnRef::hasFixedRoute (
             void  ) const
```

Returns whether the connector route is marked as fixed.

Returns

True if the connector route is fixed, false otherwise.

8.5.3.5 id()

```
unsigned int Avoid::ConnRef::id (
             void ) const
```

Returns the ID of this connector.

Returns

The ID of the connector.

8.5.3.6 needsRepaint()

```
bool Avoid::ConnRef::needsRepaint (
             void ) const
```

Returns an indication of whether this connector has a new route and thus needs to be repainted.

If the connector has been rerouted and need repainting, the
displayRoute() method can be called to get a reference to the new route.

Returns

Returns true if the connector requires repainting, or false if it does not.

8.5.3.7 route()

```
const PolyLine & Avoid::ConnRef::route (
             void ) const
```

Returns a reference to the current raw "debug" route for the connector.

This is a raw "debug" shortest path version of the route, where each line segment in the route may be made up of multiple collinear line segments. It also has no post-processing (i.e., centering, nudging apart of overlapping paths, or curving of corners) applied to it. A route to display to the user can be obtained by calling displayRoute().

Returns

The PolyLine route for the connector.

8.5.3.8 router()

```
Router * Avoid::ConnRef::router (
            void  ) const
```

Returns a pointer to the router scene this connector is in.

Returns

A pointer to the router scene for this connector.

8.5.3.9 routingCheckpoints()

```
std::vector< Checkpoint > Avoid::ConnRef::routingCheckpoints (
            void  ) const
```

Returns the current set of routing checkpoints for this connector.

Returns

The ordered list of Checkpoints that this connector will route via.

8.5.3.10 routingType()

```
ConnType Avoid::ConnRef::routingType (
            void  ) const
```

Returns the type of routing performed for this connector.

Returns

The type of routing performed.

8.5.3.11 setCallback()

```
void Avoid::ConnRef::setCallback (
            void(*)(void *) cb,
            void * ptr )
```

Sets a callback function that will called to indicate that the connector needs rerouting.

The cb function will be called when shapes are added to, removed from or moved about on the page. The pointer ptr will be passed as an argument to the callback function.

Parameters

in	*cb*	A pointer to the callback function.
in	*ptr*	A generic pointer that will be passed to the callback function.

8.5.3.12 setDestEndpoint()

```
void Avoid::ConnRef::setDestEndpoint (
             const ConnEnd & dstPoint )
```

Sets just a new destination endpoint for this connector.

If the router is using transactions, then this action will occur the next time Router::processTransaction() is called. See Router::setTransactionUse() for more information.

Parameters

in	*dstPoint*	New destination endpoint for the connector.

8.5.3.13 setEndpoints()

```
void Avoid::ConnRef::setEndpoints (
             const ConnEnd & srcPoint,
             const ConnEnd & dstPoint )
```

Sets both a new source and destination endpoint for this connector.

If the router is using transactions, then this action will occur the next time Router::processTransaction() is called. See Router::setTransactionUse() for more information.

Parameters

in	*srcPoint*	New source endpoint for the connector.
in	*dstPoint*	New destination endpoint for the connector.

8.5.3.14 setFixedExistingRoute()

```
void Avoid::ConnRef::setFixedExistingRoute (
             void  )
```

Sets a fixed existing route for this connector.

libavoid will no longer calculate object-avoiding paths for this connector but instead just return the current exisitng route.
The path of this connector will still be considered for the purpose of nudging and routing other non-fixed connectors.

Note

The endpoints of this connector will remain at their current positions, even while remaining 'attached' to shapes or junctions that move.

See also

setFixedRoute()

clearFixedRoute()

8.5.3.15 setFixedRoute()

```
void Avoid::ConnRef::setFixedRoute (
             const PolyLine & route )
```

Sets a fixed user-specified route for this connector.

libavoid will no longer calculate object-avoiding paths for this connector but instead just return the specified route. The path of this connector will still be considered for the purpose of nudging and routing other non-fixed connectors.

Note

This will reset the endpoints of the connector to the two ends of the given route, which may cause it to become dettached from any shapes or junctions. You can alternatively call setFixedExistingRoute() for connectors with valid routes in hyperedges that you would like to remain attached.

Parameters

in	*route*	The new fixed route for the connector.

See also

setFixedExistingRoute()

clearFixedRoute()

8.5.3.16 setRoutingCheckpoints()

```
void Avoid::ConnRef::setRoutingCheckpoints (
             const std::vector< Checkpoint > & checkpoints )
```

Allows the user to specify a set of checkpoints that this connector will route via.

When routing, the connector will attempt to visit each of the points in the checkpoints list in order. It will route from the source point to the first checkpoint, to the second checkpoint, etc. If a checkpoint is unreachable because it lies inside an obstacle, then that checkpoint will be skipped.

Parameters

in	*checkpoints*	An ordered list of Checkpoints that the connector will attempt to route via.

8.5.3.17 setRoutingType()

```
void Avoid::ConnRef::setRoutingType (
            ConnType type )
```

Sets the type of routing to be performed for this connector.

If a call to this method changes the current type of routing being used for the connector, then it will get rerouted during the next processTransaction() call, or immediately if transactions are not being used.

Parameters

type	The type of routing to be performed.

8.5.3.18 setSourceEndpoint()

```
void Avoid::ConnRef::setSourceEndpoint (
            const ConnEnd & srcPoint )
```

Sets just a new source endpoint for this connector.

If the router is using transactions, then this action will occur the next time Router::processTransaction() is called. See Router::setTransactionUse() for more information.

Parameters

in	*srcPoint*	New source endpoint for the connector.

8.5.3.19 splitAtSegment()

```
std::pair< JunctionRef *, ConnRef * > Avoid::ConnRef::splitAtSegment (
            const size_t segmentN )
```

Splits a connector in the centre of the segmentNth segment and creates a junction point there as well as a second connector.

The new junction and connector will be automatically added to the router scene. A slight preference will be given to the connectors connecting to the junction in the same orientation the line segment already existed in.

Returns

A pair containing pointers to the new JunctionRef and ConnRef.

The documentation for this class was generated from the following files:

- connector.h
- connector.cpp

8.6 Avoid::Edge Class Reference

A line between two points.

```
#include <geomtypes.h>
```

Public Attributes

- Point a

  The first point.
- Point b

  The second point.

8.6.1 Detailed Description

A line between two points.

The documentation for this class was generated from the following file:

- geomtypes.h

8.7 Avoid::HyperedgeNewAndDeletedObjectLists Struct Reference

The HyperedgeNewAndDeletedObjectLists class stores lists of objects created and deleted during hyperedge improvement.

```
#include <hyperedge.h>
```

Public Attributes

- JunctionRefList newJunctionList

  A list of newly created junctions.
- ConnRefList newConnectorList

  A list of newly created connectors.
- JunctionRefList deletedJunctionList

  A list of deleted junctions.
- ConnRefList deletedConnectorList

  A list of deleted connectors.
- ConnRefList changedConnectorList

  A list of changed connectors.

8.7.1 Detailed Description

The HyperedgeNewAndDeletedObjectLists class stores lists of objects created and deleted during hyperedge improvement.

After hyperedge improvement, this information can be produced by calling the Router::newAndDeletedObjectListsFromHyperedgeImpr
method.

After hyperedge rerouting, this information can be produced by calling the HyperedgeRerouter::newAndDeletedObjectLists() method for each hyperedge being fully rerouted.

The HyperedgeNewAndDeletedObjectLists::changedConnectorList attribute will only be used for hyperedge improvement and will always be empty for hyperedge rerouting.

The documentation for this struct was generated from the following file:

- hyperedge.h

8.8 Avoid::HyperedgeRerouter Class Reference

The HyperedgeRerouter class is a convenience object that can be used to register hyperedges to be rerouted, improving the placement of their junctions and connector paths.

```
#include <hyperedge.h>
```

Public Member Functions

- HyperedgeRerouter ()

    Constructor for hyperedge rerouter object.
- size_t registerHyperedgeForRerouting (ConnEndList terminals)

    Registers a hyperedge to be fully rerouted the next time the router processes a transaction.
- size_t registerHyperedgeForRerouting (JunctionRef ∗junction)

    Registers a hyperedge to be fully rerouted the next time the router processes a transaction.
- HyperedgeNewAndDeletedObjectLists newAndDeletedObjectLists (size_t index) const

    Returns a HyperedgeNewAndDeletedObjectLists detailing the lists of junctions and connectors created and deleted during hyperedge improvement.

8.8.1 Detailed Description

The HyperedgeRerouter class is a convenience object that can be used to register hyperedges to be rerouted, improving the placement of their junctions and connector paths.

To work with this class, you should get a copy from the router instance via a call to Router::hyperedgeRerouter().

If you would like a particular hyperedge to be completely rerouted with new junction positions then you should register it with this class via a call to registerHyperedgeForRerouting. A hyperedge can either be specified as a set of terminal vertices, or as a single JunctionRef. Passing a JunctionRef will cause HyperedgeRerouter to follow the attached connectors and junctions to determine the hyperedge. When you register a hyperedge you get an index number that can be used to later find information about it.

The rerouting will actually occur the next time the Router processes a transaction, see Router::processTransaction(). The rerouting will effectively create new junctions (JunctionRefs) and connectors (ConnRefs) for the hyperedge.

Since hyperedges are composed of multiple connections and junction objects, rerouting a hyperedge can cause creation of new or deletion of existing connectors and/or junctions. Thus once the transaction has been completed you should call the newAndDeletedObjectLists() to get an object containing the lists of created and deleted junctions and connectors. After the transaction You should not use references to these deleted objects any more from your own code (since the router will free their memory at its convenience) and you should refer only to the unaffected objects and the new connectors and junctions.

8.8.2 Constructor & Destructor Documentation

8.8.2.1 HyperedgeRerouter()

```
Avoid::HyperedgeRerouter::HyperedgeRerouter ( )
```

Constructor for hyperedge rerouter object.

Note

You shouldn't create this object yourself. The Router instance has one that you can request a reference to via Router::hyperedgeRerouter().

8.8.3 Member Function Documentation

8.8.3.1 newAndDeletedObjectLists()

```
HyperedgeNewAndDeletedObjectLists Avoid::HyperedgeRerouter::newAndDeletedObjectLists (
            size_t index ) const
```

Returns a HyperedgeNewAndDeletedObjectLists detailing the lists of junctions and connectors created and deleted during hyperedge improvement.

This method will only return information once the router has processed the transaction. You should read this information before processTransaction() is called again.

After calling this you should no longer refer to any of the objects in the "deleted" lists — the router will delete these and free their memory at its convenience.

Parameters

index	The index of the hyperedge to return junctions for.

Returns

A HyperedgeNewAndDeletedObjectLists containing lists of junctions and connectors created and deleted.

8.8.3.2 registerHyperedgeForRerouting() [1/2]

```
size_t Avoid::HyperedgeRerouter::registerHyperedgeForRerouting (
            ConnEndList terminals )
```

Registers a hyperedge to be fully rerouted the next time the router processes a transaction.

Parameters

in	*terminals*	The ConnEnds that form the endpoints of the hyperedge.

Returns

An index that can be used to request information on the resulting routing of the hyperedge.

8.8.3.3 registerHyperedgeForRerouting() [2/2]

```
size_t Avoid::HyperedgeRerouter::registerHyperedgeForRerouting (
            JunctionRef * junction )
```

Registers a hyperedge to be fully rerouted the next time the router processes a transaction.

In this case the connectors and junctions attached to the given junction will be traversed to determine the endpoints of the hyperedge. These endpoints will then be used for the rerouting. The junctions and connectors forming the old route will be deleted.

Parameters

in	*junction*	One of the junctions that forms the hyperedge.

Returns

An index that can be used to request information on the resulting routing of the hyperedge.

The documentation for this class was generated from the following files:

- hyperedge.h
- hyperedge.cpp

8.9 Avoid::JunctionRef Class Reference

The JunctionRef class represents a fixed or free-floating point that connectors can be attached to.

```
#include <junction.h>
```

Inherits Avoid::Obstacle.

Public Member Functions

- JunctionRef (Router ∗router, Point position, const unsigned int id=0)

  Junction reference constructor.
- virtual ∼JunctionRef ()

  Junction reference destructor.
- ConnRef ∗ removeJunctionAndMergeConnectors (void)

  Removes a junction that has only two connectors attached to it and merges them into a single connector.
- Point position (void) const

  Returns the position of this junction.
- void setPositionFixed (bool fixed)

  Sets whether the junction has a fixed position and therefore can't be moved by the Router during routing.
- bool positionFixed (void) const

  Returns whether this junction has a fixed position (that can't be moved by the Router during routing).
- Point recommendedPosition (void) const

  Returns a recommended position for the junction based on improving hyperedge routes. This value will be set during routing when the improveHyperedgeRoutesMovingJunctions router option is set (the default).
- unsigned int **id** (void) const

  Returns the ID of this obstacle.
- const Polygon & **polygon** (void) const

  Returns a reference to the polygon boundary of this obstacle.
- Router ∗ **router** (void) const

  Returns a pointer to the router scene this obstacle is in.

8.9.1 Detailed Description

The JunctionRef class represents a fixed or free-floating point that connectors can be attached to.

A JunctionRef represents a junction between multiple connectors, or could be used to specify an intermediate point that a single connector must route through.

8.9.2 Constructor & Destructor Documentation

8.9.2.1 JunctionRef()

```
Avoid::JunctionRef::JunctionRef (
             Router * router,
             Point position,
             const unsigned int id = 0 )
```

Junction reference constructor.

Creates a junction object reference, and adds it to the router scene. This junction will be considered to be an obstacle. This will cause connectors intersecting the newly added junction to be marked as needing to be rerouted.

If the router is using transactions, then changes will occur the next time Router::processTransaction() is called. See Router::setTransactionUse() for more information.

The junction can be moved with Router::moveJunction() and removed from the scene and freed with Router::deleteJunction().

libavoid expects junctions to have sensible positions (i.e., for junctions to be positioned outside of shapes). When routing it will simplify hyperedges by moving junctions while preserving hyperedge topology, i.e., not altering the sides of shapes the hyperedge routes around.

If you don't have sensible positions for junctions or want to disregard the junction position and reroute the entire hyperedge considering only the endpoints, then this can be achieved by registering the hyperedge with the HyperedgeRerouter class obtained by calling the Router::hyperedgeRerouter() method.

When the improveHyperedgeRoutesMovingJunctions router option is set (the default) the junction position is a suggestion used for initial routing, but subsequent hyperedge path improvement may suggest new junction positions for the updated routings. This position can be accessed via the recommendedPosition() method.

When the improveHyperedgeRoutesMovingAddingAndDeletingJunctions router option is set (not the default) junctions and connectors can be added or removed to further improve hyperedges, see also Router::newAndDeletedObjectListsFromHyperedgeImprovement().

Note

Regarding IDs: You can let libavoid manually handle IDs by not specifying them. Alternatively, you can specify all IDs yourself, but you must be careful to makes sure that each object in the scene (shape, connector, cluster, etc.) is given a unique, positive ID. This uniqueness is checked if assertions are enabled, but if not and there are clashes then strange things can happen.

Parameters

`in`	*router*	The router scene to place the junction into.
`in`	*position*	A Point representing the position of the junction.
`in`	*id*	Optionally, a positive integer ID unique among all objects.

8.9.2.2 ∼JunctionRef()

```
Avoid::JunctionRef::∼JunctionRef ( )  [virtual]
```

Junction reference destructor.

Do not call this yourself, instead call Router::deleteJunction(). Ownership of this object belongs to the router scene.

8.9.3 Member Function Documentation

8.9.3.1 id()

```
unsigned int Avoid::Obstacle::id (
            void ) const [inherited]
```

Returns the ID of this obstacle.

Returns

The ID of the obstacle.

8.9.3.2 polygon()

```
const Polygon & Avoid::Obstacle::polygon (
            void ) const [inherited]
```

Returns a reference to the polygon boundary of this obstacle.

Returns

A reference to the polygon boundary of the obstacle.

8.9.3.3 position()

```
Point Avoid::JunctionRef::position (
            void ) const
```

Returns the position of this junction.

Returns

A point representing the position of this junction.

8.9.3.4 positionFixed()

```
bool Avoid::JunctionRef::positionFixed (
            void ) const
```

Returns whether this junction has a fixed position (that can't be moved by the Router during routing).

Returns

A point representing the position of this junction.

8.9.3.5 recommendedPosition()

```
Point Avoid::JunctionRef::recommendedPosition (
            void  ) const
```

Returns a recommended position for the junction based on improving hyperedge routes. This value will be set during routing when the improveHyperedgeRoutesMovingJunctions router option is set (the default).

Returns

A point indicating the ideal position for this junction.

8.9.3.6 removeJunctionAndMergeConnectors()

```
ConnRef * Avoid::JunctionRef::removeJunctionAndMergeConnectors (
            void  )
```

Removes a junction that has only two connectors attached to it and merges them into a single connector.

The junction and one of the connectors will be removed from the router scene and the connector deleted. A pointer to the remaining (merged) connector will be returned by this method.

Currently this method does not delete and free the Junction itself. The user needs to do this after the transaction has been processed by the router.

If there are more than two connectors attached to the junction then nothing will be changed and this method will return nullptr.

Returns

The merged connector, or nullptr if the junction was not removed.

8.9.3.7 router()

```
Router * Avoid::Obstacle::router (
            void  ) const  [inherited]
```

Returns a pointer to the router scene this obstacle is in.

Returns

A pointer to the router scene for this obstacle.

8.9.3.8 setPositionFixed()

```
void Avoid::JunctionRef::setPositionFixed (
            bool fixed )
```

Sets whether the junction has a fixed position and therefore can't be moved by the Router during routing.

This property is ignored for hyperedge improvement if the option improveHyperedgeRoutesMovingAddingAnd↵
DeletingJunctions is set and when it would lead to confusing hyperedge topology, such as two overlapping junctions with a zero length connector between them or an unnecessary junction bridging two connectors.

Parameters

in	*fixed*	Boolean indicating whether the junction position should be marked as fixed.

The documentation for this class was generated from the following files:

- junction.h
- junction.cpp

8.10 Avoid::Point Class Reference

The Point class defines a point in the plane.

```
#include <geomtypes.h>
```

Public Member Functions

- Point ()

  Default constructor.
- Point (const double xv, const double yv)

  Standard constructor.
- bool operator== (const Point &rhs) const

  Comparison operator. Returns true if at same position.
- bool equals (const Point &rhs, double epsilon=0.0001) const

  Comparison operator. Returns true if at same position, or at effectively the same position for a given value of epsilson.
- bool operator!= (const Point &rhs) const

  Comparison operator. Returns true if at different positions.
- bool operator< (const Point &rhs) const

  Comparison operator. Returns true if less-then rhs point.
- double & operator[ ] (const size_t dimension)

  Returns the x or y value of the point, given the dimension.

Public Attributes

- double x

  The x position.
- double y

  The y position.
- unsigned int id

  The ID associated with this point.
- unsigned short vn

  The vertex number associated with this point.

8.10.1 Detailed Description

The Point class defines a point in the plane.

Points consist of an x and y value. They may also have an ID and vertex number associated with them.

8.10.2 Constructor & Destructor Documentation

8.10.2.1 Point()

```
Avoid::Point::Point (
            const double xv,
            const double yv )
```

Standard constructor.

Parameters

in	*xv*	The x position of the point.
in	*yv*	The y position of the point.

8.10.3 Member Function Documentation

8.10.3.1 equals()

```
bool Avoid::Point::equals (
            const Point & rhs,
            double epsilon = 0.0001 ) const
```

Comparison operator. Returns true if at same position, or at effectively the same position for a given value of epsilson.

Parameters

in	*rhs*	The point to compare with this one.
in	*epsilon*	Value of epsilon to use during comparison.

Returns

The result of the comparison.

See also

operator==()

8.10.3.2 operator"!=()

```
bool Avoid::Point::operator!= (
             const Point & rhs ) const
```

Comparison operator. Returns true if at different positions.

Parameters

in	*rhs*	The point to compare with this one.

Returns

The result of the comparison.

8.10.3.3 operator<()

```
bool Avoid::Point::operator< (
             const Point & rhs ) const
```

Comparison operator. Returns true if less-then rhs point.

Note

This operator is not particularly useful, but is defined to allow std::set<Point>.

Parameters

in	*rhs*	The point to compare with this one.

Returns

The result of the comparison.

8.10.3.4 operator==()

```
bool Avoid::Point::operator== (
             const Point & rhs ) const
```

Comparison operator. Returns true if at same position.

Parameters

in	*rhs*	The point to compare with this one.

Returns

The result of the comparison.

See also

equals()

8.10.3.5 operator[]()

```
double & Avoid::Point::operator[ ] (
             const size_t dimension )
```

Returns the x or y value of the point, given the dimension.

Parameters

in	*dimension*	The dimension: 0 for x, 1 for y.

Returns

The component of the point in that dimension.

The documentation for this class was generated from the following files:

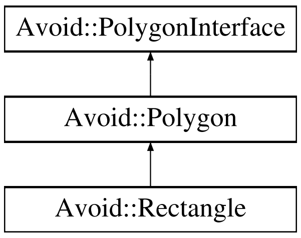
- geomtypes.h
- geomtypes.cpp

8.11 Avoid::Polygon Class Reference

A dynamic Polygon, to which points can be easily added and removed.

```
#include <geomtypes.h>
```

Inheritance diagram for Avoid::Polygon:



Public Member Functions

- Polygon ()

  Constructs an empty polygon (with zero points).
- Polygon (const int n)

  Constructs a new polygon with n points.
- Polygon (const PolygonInterface &poly)

  Constructs a new polygon from an existing Polygon.
- void clear (void)

  Resets this to the empty polygon.
- bool empty (void) const

  Returns true if this polygon is empty.
- size_t size (void) const

  Returns the number of points in this polygon.
- int id (void) const

  Returns the ID value associated with this polygon.
- const Point & at (size_t index) const

  Returns a specific point in the polygon.
- void setPoint (size_t index, const Point &point)

  Sets a position for a particular point in the polygon..
- Polygon simplify (void) const

  Returns a simplified Polyline, where all collinear line segments have been collapsed down into single line segments.
- Polygon curvedPolyline (const double curve_amount, const bool closed=false) const

  Returns a curved approximation of this multi-segment PolyLine, with the corners replaced by smooth Bezier curves.
- void translate (const double xDist, const double yDist)

  Translates the polygon position by a relative amount.
- Polygon boundingRectPolygon (void) const

  Returns the bounding rectangle for this polygon.
- Box offsetBoundingBox (double offset) const

  Returns the bounding rectangle that contains this polygon with optionally some buffer space around it for routing.

Public Attributes

- int _id

  An ID for the polygon.
- std::vector< Point > ps

  A vector of the points that make up the Polygon.
- std::vector< char > ts

  If used, denotes whether the corresponding point in ps is a move-to operation or a Bezier curve-to.

8.11.1 Detailed Description

A dynamic Polygon, to which points can be easily added and removed.

Note

The Rectangle class can be used as an easy way of constructing a square or rectangular polygon.

8.11.2 Constructor & Destructor Documentation

8.11.2.1 Polygon() [1/2]

```
Avoid::Polygon::Polygon (
            const int n )
```

Constructs a new polygon with n points.

A rectangle would be comprised of four point. An n segment PolyLine (represented as a Polygon) would be comprised of n+1 points. Whether a particular Polygon is closed or not, depends on whether it is a Polygon or Polyline. Shape polygons are always considered to be closed, meaning the last point joins back to the first point.

The values for points can be set by setting the Polygon:ps vector, or via the Polygon::setPoint() method.

Parameters

in	*n*	Number of points in the polygon.

8.11.2.2 Polygon() [2/2]

```
Avoid::Polygon::Polygon (
            const PolygonInterface & poly )
```

Constructs a new polygon from an existing Polygon.

Parameters

in	*poly*	An existing polygon to copy the new polygon from.

8.11.3 Member Function Documentation

8.11.3.1 at()

```
const Point & Avoid::Polygon::at (
            size_t index ) const  [virtual]
```

Returns a specific point in the polygon.

Parameters

in	*index*	The array index of the point to be returned.

Implements Avoid::PolygonInterface.

8.11.3.2 boundingRectPolygon()

```
Polygon Avoid::PolygonInterface::boundingRectPolygon (
             void  ) const  [inherited]
```

Returns the bounding rectangle for this polygon.

Returns

A new Rectangle representing the bounding box.

8.11.3.3 curvedPolyline()

```
Polygon Avoid::Polygon::curvedPolyline (
             const double curve_amount,
             const bool closed = false ) const
```

Returns a curved approximation of this multi-segment PolyLine, with the corners replaced by smooth Bezier curves.

This function does not do any further obstacle avoidance with the curves produced. Hence, you would usually specify a curve_amount in similar size to the space buffer around obstacles in the scene. This way the curves will cut the corners around shapes but still run within this buffer space.

Parameters

curve_amount	Describes the distance along the end of each line segment to turn into a curve.
closed	Describes whether the Polygon should be treated as closed. Defaults to false.

Returns

A new polyline (polygon) representing the curved path. Its points represent endpoints of line segments and Bezier spline control points. The Polygon::ts vector for this returned polygon is populated with a character for each point describing its type.

See also

ts

8.11.3.4 offsetBoundingBox()

```
Box Avoid::PolygonInterface::offsetBoundingBox (
             double offset ) const  [inherited]
```

Returns the bounding rectangle that contains this polygon with optionally some buffer space around it for routing.

If a buffer distance of zero is given, then this method returns the bounding rectangle for the shape's polygon.

Parameters

offset	Extra distance to pad each side of the rect.

Returns

The bounding box for the polygon.

8.11.3.5 setPoint()

```
void Avoid::Polygon::setPoint (
             size_t index,
             const Point & point )
```

Sets a position for a particular point in the polygon..

Parameters

in	*index*	The array index of the point to be set.
in	*point*	The point value to be assigned..

8.11.3.6 simplify()

```
Polygon Avoid::Polygon::simplify (
             void  ) const
```

Returns a simplified Polyline, where all collinear line segments have been collapsed down into single line segments.

Returns

A new polyline with a simplified representation.

8.11.3.7 translate()

```
void Avoid::Polygon::translate (
             const double xDist,
             const double yDist )
```

Translates the polygon position by a relative amount.

Parameters

`in`	*xDist*	Distance to move polygon in the x dimension.
`in`	*yDist*	Distance to move polygon in the y dimension.

8.11.4 Member Data Documentation

8.11.4.1 ts

```
std::vector<char> Avoid::Polygon::ts
```

If used, denotes whether the corresponding point in ps is a move-to operation or a Bezier curve-to.

Each character describes the drawing operation for the corresponding point in the ps vector. Possible values are:

- 'M': A moveto operation, marks the first point;
- 'L': A lineto operation, is a line from the previous point to the current point; or
- 'C': A curveto operation, three consecutive 'C' points (along with the previous point) describe the control points of a Bezier curve.
- 'Z': Closes the path (used for cluster boundaries).

Note

This vector will currently only be populated for polygons returned by curvedPolyline().

The documentation for this class was generated from the following files:

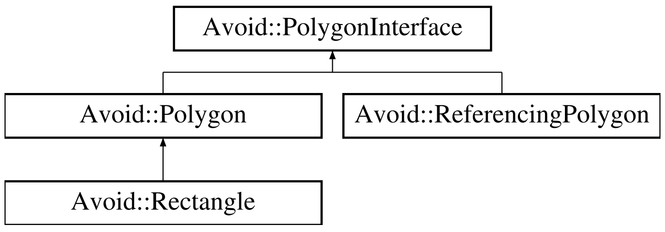
- geomtypes.h
- geomtypes.cpp

8.12 Avoid::PolygonInterface Class Reference

A common interface used by the Polygon classes.

```
#include <geomtypes.h>
```

Inheritance diagram for Avoid::PolygonInterface:



Public Member Functions

- PolygonInterface ()

    Constructor.
- virtual ∼PolygonInterface ()

    Destructor.
- virtual void clear (void)=0

    Resets this to the empty polygon.
- virtual bool empty (void) const =0

    Returns true if this polygon is empty.
- virtual size_t size (void) const =0

    Returns the number of points in this polygon.
- virtual int id (void) const =0

    Returns the ID value associated with this polygon.
- virtual const Point & at (size_t index) const =0

    Returns a specific point in the polygon.
- Polygon boundingRectPolygon (void) const

    Returns the bounding rectangle for this polygon.
- Box offsetBoundingBox (double offset) const

    Returns the bounding rectangle that contains this polygon with optionally some buffer space around it for routing.

8.12.1 Detailed Description

A common interface used by the Polygon classes.

8.12.2 Member Function Documentation

8.12.2.1 at()

```
virtual const Point& Avoid::PolygonInterface::at (
            size_t index ) const  [pure virtual]
```

Returns a specific point in the polygon.

Parameters

in	*index*	The array index of the point to be returned.

Implemented in Avoid::ReferencingPolygon, and Avoid::Polygon.

8.12.2.2 boundingRectPolygon()

```
Polygon Avoid::PolygonInterface::boundingRectPolygon (
            void  ) const
```

Returns the bounding rectangle for this polygon.

Returns

A new Rectangle representing the bounding box.

8.12.2.3 offsetBoundingBox()

```
Box Avoid::PolygonInterface::offsetBoundingBox (
             double offset ) const
```

Returns the bounding rectangle that contains this polygon with optionally some buffer space around it for routing.

If a buffer distance of zero is given, then this method returns the bounding rectangle for the shape's polygon.

Parameters

offset	Extra distance to pad each side of the rect.

Returns

The bounding box for the polygon.

The documentation for this class was generated from the following files:

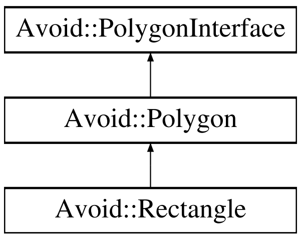
- geomtypes.h
- geomtypes.cpp

8.13 Avoid::Rectangle Class Reference

A Rectangle, a simpler way to define the polygon for square or rectangular shapes.

```
#include <geomtypes.h>
```

Inheritance diagram for Avoid::Rectangle:



Public Member Functions

- Rectangle (const Point &topLeft, const Point &bottomRight)

  Constructs a rectangular polygon given two opposing corner points.
- Rectangle (const Point &centre, const double width, const double height)

  Constructs a rectangular polygon given the centre, width and height.
- void clear (void)

  Resets this to the empty polygon.
- bool empty (void) const

  Returns true if this polygon is empty.
- size_t size (void) const

  Returns the number of points in this polygon.
- int id (void) const

  Returns the ID value associated with this polygon.
- const Point & at (size_t index) const

  Returns a specific point in the polygon.
- void setPoint (size_t index, const Point &point)

  Sets a position for a particular point in the polygon..
- Polygon simplify (void) const

  Returns a simplified Polyline, where all collinear line segments have been collapsed down into single line segments.
- Polygon curvedPolyline (const double curve_amount, const bool closed=false) const

  Returns a curved approximation of this multi-segment PolyLine, with the corners replaced by smooth Bezier curves.
- void translate (const double xDist, const double yDist)

  Translates the polygon position by a relative amount.
- Polygon boundingRectPolygon (void) const

  Returns the bounding rectangle for this polygon.
- Box offsetBoundingBox (double offset) const

  Returns the bounding rectangle that contains this polygon with optionally some buffer space around it for routing.

Public Attributes

- int _id

  An ID for the polygon.
- std::vector< Point > ps

  A vector of the points that make up the Polygon.
- std::vector< char > ts

  If used, denotes whether the corresponding point in ps is a move-to operation or a Bezier curve-to.

8.13.1 Detailed Description

A Rectangle, a simpler way to define the polygon for square or rectangular shapes.

8.13.2 Constructor & Destructor Documentation

8.13.2.1 Rectangle() [1/2]

```
Avoid::Rectangle::Rectangle (
            const Point & topLeft,
            const Point & bottomRight )
```

Constructs a rectangular polygon given two opposing corner points.

Parameters

in	*topLeft*	The first corner point of the rectangle.
in	*bottomRight*	The opposing corner point of the rectangle.

8.13.2.2 Rectangle() [2/2]

```
Avoid::Rectangle::Rectangle (
            const Point & centre,
            const double width,
            const double height )
```

Constructs a rectangular polygon given the centre, width and height.

Parameters

in	*centre*	The centre of the rectangle, specified as a point.
in	*width*	The width of the rectangle.
in	*height*	The height of the rectangle.

8.13.3 Member Function Documentation

8.13.3.1 at()

```
const Point & Avoid::Polygon::at (
            size_t index ) const [virtual], [inherited]
```

Returns a specific point in the polygon.

Parameters

in	*index*	The array index of the point to be returned.

Implements Avoid::PolygonInterface.

8.13.3.2 boundingRectPolygon()

```
Polygon Avoid::PolygonInterface::boundingRectPolygon (
            void ) const [inherited]
```

Returns the bounding rectangle for this polygon.

Returns

A new Rectangle representing the bounding box.

8.13.3.3 curvedPolyline()

```
Polygon Avoid::Polygon::curvedPolyline (
             const double curve_amount,
             const bool closed = false ) const  [inherited]
```

Returns a curved approximation of this multi-segment PolyLine, with the corners replaced by smooth Bezier curves.

This function does not do any further obstacle avoidance with the curves produced. Hence, you would usually specify a curve_amount in similar size to the space buffer around obstacles in the scene. This way the curves will cut the corners around shapes but still run within this buffer space.

Parameters

curve_amount	Describes the distance along the end of each line segment to turn into a curve.
closed	Describes whether the Polygon should be treated as closed. Defaults to false.

Returns

A new polyline (polygon) representing the curved path. Its points represent endpoints of line segments and Bezier spline control points. The Polygon::ts vector for this returned polygon is populated with a character for each point describing its type.

See also

ts

8.13.3.4 offsetBoundingBox()

```
Box Avoid::PolygonInterface::offsetBoundingBox (
             double offset ) const  [inherited]
```

Returns the bounding rectangle that contains this polygon with optionally some buffer space around it for routing.

If a buffer distance of zero is given, then this method returns the bounding rectangle for the shape's polygon.

Parameters

offset	Extra distance to pad each side of the rect.

Returns

The bounding box for the polygon.

8.13.3.5 setPoint()

```
void Avoid::Polygon::setPoint (
             size_t index,
             const Point & point )  [inherited]
```

Sets a position for a particular point in the polygon..

Parameters

in	*index*	The array index of the point to be set.
in	*point*	The point value to be assigned..

8.13.3.6 simplify()

```
Polygon Avoid::Polygon::simplify (
             void  ) const  [inherited]
```

Returns a simplified Polyline, where all collinear line segments have been collapsed down into single line segments.

Returns

A new polyline with a simplified representation.

8.13.3.7 translate()

```
void Avoid::Polygon::translate (
             const double xDist,
             const double yDist )  [inherited]
```

Translates the polygon position by a relative amount.

Parameters

in	*xDist*	Distance to move polygon in the x dimension.
in	*yDist*	Distance to move polygon in the y dimension.

8.13.4 Member Data Documentation

8.13.4.1 ts

```
std::vector<char> Avoid::Polygon::ts [inherited]
```

If used, denotes whether the corresponding point in ps is a move-to operation or a Bezier curve-to.

Each character describes the drawing operation for the corresponding point in the ps vector. Possible values are:

- 'M': A moveto operation, marks the first point;
- 'L': A lineto operation, is a line from the previous point to the current point; or
- 'C': A curveto operation, three consecutive 'C' points (along with the previous point) describe the control points of a Bezier curve.
- 'Z': Closes the path (used for cluster boundaries).

Note

This vector will currently only be populated for polygons returned by curvedPolyline().

The documentation for this class was generated from the following files:

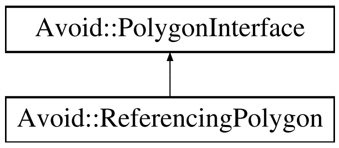
- geomtypes.h
- geomtypes.cpp

8.14 Avoid::ReferencingPolygon Class Reference

A Polygon which just references its points from other Polygons.

```
#include <geomtypes.h>
```

Inheritance diagram for Avoid::ReferencingPolygon:



Public Member Functions

- void clear (void)
  Resets this to the empty polygon.
- bool empty (void) const
  Returns true if this polygon is empty.
- size_t size (void) const
  Returns the number of points in this polygon.
- int id (void) const
  Returns the ID value associated with this polygon.
- const Point & at (size_t index) const
  Returns a specific point in the polygon.
- Polygon boundingRectPolygon (void) const
  Returns the bounding rectangle for this polygon.
- Box offsetBoundingBox (double offset) const
  Returns the bounding rectangle that contains this polygon with optionally some buffer space around it for routing.

8.14.1 Detailed Description

A Polygon which just references its points from other Polygons.

This type of Polygon is used to accurately represent cluster boundaries made up from the corner points of shapes.

8.14.2 Member Function Documentation

8.14.2.1 at()

```
const Point & Avoid::ReferencingPolygon::at (
             size_t index ) const  [virtual]
```

Returns a specific point in the polygon.

Parameters

in	*index*	The array index of the point to be returned.

Implements Avoid::PolygonInterface.

8.14.2.2 boundingRectPolygon()

```
Polygon Avoid::PolygonInterface::boundingRectPolygon (
             void  ) const  [inherited]
```

Returns the bounding rectangle for this polygon.

Returns

A new Rectangle representing the bounding box.

8.14.2.3 offsetBoundingBox()

```
Box Avoid::PolygonInterface::offsetBoundingBox (
             double offset ) const  [inherited]
```

Returns the bounding rectangle that contains this polygon with optionally some buffer space around it for routing.

If a buffer distance of zero is given, then this method returns the bounding rectangle for the shape's polygon.

Parameters

offset	Extra distance to pad each side of the rect.

Returns

The bounding box for the polygon.

The documentation for this class was generated from the following files:

- geomtypes.h
- geomtypes.cpp

8.15 Avoid::Router Class Reference

The Router class represents a libavoid router instance.

```
#include <router.h>
```

Public Member Functions

- Router (const unsigned int flags)

  Constructor for router instance.
- virtual ∼Router ()

  Destructor for router instance.
- void setTransactionUse (const bool transactions)

  Allows setting of the behaviour of the router in regard to transactions. This controls whether transactions are used to queue changes and process them efficiently at once or they are instead processed immediately.
- bool transactionUse (void) const

  Reports whether the router groups actions into transactions.
- bool processTransaction (void)

  Finishes the current transaction and processes all the queued object changes efficiently.
- void deleteShape (ShapeRef ∗shape)

Delete a shape from the router scene.
- void moveShape (ShapeRef ∗shape, const Polygon &newPoly, const bool first_move=false)

  Move or resize an existing shape within the router scene.
- void moveShape (ShapeRef ∗shape, const double xDiff, const double yDiff)

  Move an existing shape within the router scene by a relative distance.
- void deleteJunction (JunctionRef ∗junction)

  Remove a junction from the router scene.
- void deleteConnector (ConnRef ∗connector)

  Remove a connector from the router scene.
- void moveJunction (JunctionRef ∗junction, const Point &newPosition)

  Move an existing junction within the router scene.
- void moveJunction (JunctionRef ∗junction, const double xDiff, const double yDiff)

  Move an existing junction within the router scene by a relative distance.
- void setRoutingParameter (const RoutingParameter parameter, const double value=chooseSensibleParam←Value)

  Sets values for routing parameters, including routing penalties.
- double routingParameter (const RoutingParameter parameter) const

  Returns the current value for a particular routing parameter of a given type.
- void setRoutingOption (const RoutingOption option, const bool value)

  Turn specific routing options on or off.
- bool routingOption (const RoutingOption option) const

  Returns the current state for a specific routing option.
- void setRoutingPenalty (const RoutingParameter penType, const double penVal=chooseSensibleParam←Value)

  Sets or removes penalty values that are applied during connector routing.
- HyperedgeRerouter ∗ hyperedgeRerouter (void)

  Returns a pointer to the hyperedge rerouter for the router.
- void outputInstanceToSVG (std::string filename=std::string())

  Generates an SVG file containing debug output and code that can be used to regenerate the instance.
- virtual unsigned int newObjectId (void) const

  Returns the object ID used for automatically generated objects, such as during hyperedge routing.
- bool objectIdIsUnused (const unsigned int id) const

  Returns whether or not the given ID is already used.
- virtual bool shouldContinueTransactionWithProgress (unsigned int elapsedTime, unsigned int phaseNumber, unsigned int totalPhases, double proportion)

  A method called at regular intervals during transaction processing to report progress and ask if the Router should continue the transaction.
- HyperedgeNewAndDeletedObjectLists newAndDeletedObjectListsFromHyperedgeImprovement (void) const

  Returns a HyperedgeNewAndDeletedObjectLists detailing the lists of junctions and connectors created and deleted during hyperedge improvement.
- void setTopologyAddon (TopologyAddonInterface ∗topologyAddon)

  Set an addon for doing orthogonal topology improvement.

8.15.1 Detailed Description

The Router class represents a libavoid router instance.

Usually you would keep a separate Router instance for each diagram or layout you have open in your application.

8.15.2 Constructor & Destructor Documentation

8.15.2.1 Router()

```
Avoid::Router::Router (
            const unsigned int flags )
```

Constructor for router instance.

Parameters

in	*flags*	One or more Avoid::RouterFlag options to control the behaviour of the router.

8.15.2.2 ∼Router()

```
Avoid::Router::∼Router ( ) [virtual]
```

Destructor for router instance.

Note

Destroying a router instance will delete all remaining shapes and connectors, thereby invalidating any existing pointers to them.

8.15.3 Member Function Documentation

8.15.3.1 deleteConnector()

```
void Avoid::Router::deleteConnector (
            ConnRef * connector )
```

Remove a connector from the router scene.

If the router is using transactions, then this action will occur the next time Router::processTransaction() is called. See Router::setTransactionUse() for more information.

You should not use the connector reference again after this call. The router will handle freeing of the connector's memory.

Parameters

in	*connector*	Pointer reference to the connector being removed.

8.15.3.2 deleteJunction()

```
void Avoid::Router::deleteJunction (
             JunctionRef * junction )
```

Remove a junction from the router scene.

If the router is using transactions, then this action will occur the next time Router::processTransaction() is called. See Router::setTransactionUse() for more information.

You should not use the junction reference again after this call. The router will handle freeing of the junction's memory.

Parameters

in	*junction*	Pointer reference to the junction being removed.

8.15.3.3 deleteShape()

```
void Avoid::Router::deleteShape (
             ShapeRef * shape )
```

Delete a shape from the router scene.

Connectors that could have a better (usually shorter) path after the removal of this shape will be marked as needing to be rerouted.

If the router is using transactions, then this action will occur the next time Router::processTransaction() is called. See Router::setTransactionUse() for more information.

You should not use the shape reference again after this call. The router will handle freeing of the shape's memory.

Parameters

in	*shape*	Pointer reference to the shape being removed.

8.15.3.4 hyperedgeRerouter()

```
HyperedgeRerouter * Avoid::Router::hyperedgeRerouter (
             void )
```

Returns a pointer to the hyperedge rerouter for the router.

Returns

A HyperedgeRerouter object that can be used to register hyperedges for rerouting.

8.15.3.5 moveJunction() `[1/2]`

```
void Avoid::Router::moveJunction (
             JunctionRef * junction,
             const double xDiff,
             const double yDiff )
```

Move an existing junction within the router scene by a relative distance.

Connectors that are attached to this junction will be rerouted as a result of the move.

If the router is using transactions, then this action will occur the next time Router::processTransaction() is called. See Router::setTransactionUse() for more information.

Parameters

in	*junction*	Pointer reference to the junction being moved.
in	*xDiff*	The distance to move the junction in the x dimension.
in	*yDiff*	The distance to move the junction in the y dimension.

8.15.3.6 moveJunction() `[2/2]`

```
void Avoid::Router::moveJunction (
             JunctionRef * junction,
             const Point & newPosition )
```

Move an existing junction within the router scene.

Connectors that are attached to this junction will be rerouted as a result of the move.

If the router is using transactions, then this action will occur the next time Router::processTransaction() is called. See Router::setTransactionUse() for more information.

Parameters

in	*junction*	Pointer reference to the junction being moved.
in	*newPosition*	The new position for the junction.

8.15.3.7 moveShape() `[1/2]`

```
void Avoid::Router::moveShape (
            ShapeRef * shape,
            const double xDiff,
            const double yDiff )
```

Move an existing shape within the router scene by a relative distance.

Connectors that intersect the shape's new position, or that could have a better (usually shorter) path after the change, will be marked as needing to be rerouted.

If the router is using transactions, then this action will occur the next time Router::processTransaction() is called. See Router::setTransactionUse() for more information.

Parameters

in	*shape*	Pointer reference to the shape being moved.
in	*xDiff*	The distance to move the shape in the x dimension.
in	*yDiff*	The distance to move the shape in the y dimension.

8.15.3.8 moveShape() [2/2]

```
void Avoid::Router::moveShape (
            ShapeRef * shape,
            const Polygon & newPoly,
            const bool first_move = false )
```

Move or resize an existing shape within the router scene.

A new polygon for the shape can be given to effectively move or resize the shape with the scene. Connectors that intersect the new shape polygon, or that could have a better (usually shorter) path after the change, will be marked as needing to be rerouted.

If the router is using transactions, then this action will occur the next time Router::processTransaction() is called. See Router::setTransactionUse() for more information.

Parameters

in	*shape*	Pointer reference to the shape being moved/resized.
in	*newPoly*	The new polygon boundary for the shape.
in	*first_move*	This option is used for some advanced (currently undocumented) behaviour and it should be ignored for the moment.

8.15.3.9 newAndDeletedObjectListsFromHyperedgeImprovement()

```
HyperedgeNewAndDeletedObjectLists Avoid::Router::newAndDeletedObjectListsFromHyperedgeImprovement
(
            void ) const
```

Returns a HyperedgeNewAndDeletedObjectLists detailing the lists of junctions and connectors created and deleted during hyperedge improvement.

This method will only return information once the router has processed the transaction. You should read and act on this information before processTransaction() is called again.

After calling this you should no longer refer to any of the objects in the "deleted" lists — the router will delete these and free their memory at its convenience.

Returns

A HyperedgeNewAndDeletedObjectLists containing lists of junctions and connectors created and deleted.

8.15.3.10 newObjectId()

```
unsigned int Avoid::Router::newObjectId (
             void ) const [virtual]
```

Returns the object ID used for automatically generated objects, such as during hyperedge routing.

Reimplement this in a subclass to set specific IDs for new objects.

Note

Your implementation should return a value that does not fail objectIdIsUnused().

Returns

The ID for a new object.

8.15.3.11 objectIdIsUnused()

```
bool Avoid::Router::objectIdIsUnused (
             const unsigned int id ) const
```

Returns whether or not the given ID is already used.

You should only need this if you reimplement newObjectId().

Parameters

in	*id*	An ID to test.

Returns

A boolean denoting that the given ID is unused.

8.15.3.12 outputInstanceToSVG()

```
void Avoid::Router::outputInstanceToSVG (
             std::string filename = std::string() )
```

Generates an SVG file containing debug output and code that can be used to regenerate the instance.

If transactions are being used, then this method should be called after processTransaction() has been called, so that it includes any changes being queued by the router.

Parameters

in	*filename*	A string indicating the filename (without extension) for the output file. Defaults to "libavoid-debug.svg" if no filename is given.

8.15.3.13 processTransaction()

```
bool Avoid::Router::processTransaction (
             void )
```

Finishes the current transaction and processes all the queued object changes efficiently.

This method will efficiently process all moves, additions and deletions that have occurred since processTransaction() was last called.

If transactionUse() is false, then all actions will have been processed immediately and this method will do nothing.

Returns

A boolean value describing whether there were any actions to process.

See also

setTransactionUse

8.15.3.14 routingOption()

```
bool Avoid::Router::routingOption (
             const RoutingOption option ) const
```

Returns the current state for a specific routing option.

Parameters

in	*option*	The type of routing option, a RoutingOption.

Returns

A boolean representing the option state.

8.15.3.15 routingParameter()

```
double Avoid::Router::routingParameter (
            const RoutingParameter parameter ) const
```

Returns the current value for a particular routing parameter of a given type.

Parameters

in	*parameter*	The type of parameter, a RoutingParameter.

Returns

The value for the specified routing parameter.

8.15.3.16 setRoutingOption()

```
void Avoid::Router::setRoutingOption (
            const RoutingOption option,
            const bool value )
```

Turn specific routing options on or off.

Parameters

in	*option*	The type of routing option, a RoutingOption.
in	*value*	A boolean representing the option state.

8.15.3.17 setRoutingParameter()

```
void Avoid::Router::setRoutingParameter (
            const RoutingParameter parameter,
            const double value = chooseSensibleParamValue )
```

Sets values for routing parameters, including routing penalties.

libavoid uses a set of parameters to allow the user more control over routing style and quality. These different parameters are described and explained by the RoutingParameter enum. All parameters have sensible defaults.

Regarding routing penalties, libavoid will by default produce shortest path routes between the source and destination points for each connector. There are several penalties that can be applied during this stage to penalise certain conditions and thus improve the aesthetics of the routes generated.

If a value of zero or Avoid::zeroParamValue is given then the particular parameter value or penalty will be removed. If no parameter value argument (or a negative value) is specified when calling this method, then a sensible penalty value will be automatically chosen.

This method does not re-trigger processing of connectors. The new parameter value will be used the next time rerouting is performed.

Parameters

in	*parameter*	The type of penalty, a RoutingParameter.
in	*value*	The value to be set for that parameter.

8.15.3.18 setRoutingPenalty()

```
void Avoid::Router::setRoutingPenalty (
            const RoutingParameter penType,
            const double penVal = chooseSensibleParamValue )
```

Sets or removes penalty values that are applied during connector routing.

Note

This is a convenience wrapper for the setRoutingParameter()

Parameters

in	*penType*	The type of penalty, a RoutingParameter.
in	*penVal*	The value to be applied for each occurrence of the penalty case.

8.15.3.19 setTopologyAddon()

```
void Avoid::Router::setTopologyAddon (
            TopologyAddonInterface * topologyAddon )
```

Set an addon for doing orthogonal topology improvement.

It is expected that you would use the topology::AvoidTopologyAddon() from libtopology rather than write your own. This is done so that libavoid does not have to depend on libtopology.

8.15.3.20 setTransactionUse()

```
void Avoid::Router::setTransactionUse (
            const bool transactions )
```

Allows setting of the behaviour of the router in regard to transactions. This controls whether transactions are used to queue changes and process them efficiently at once or they are instead processed immediately.

It is more efficient to perform actions like shape movement, addition or deletion as batch tasks, and reroute the necessary connectors just once after these actions have been performed. For this reason, libavoid allows you to group such actions into "transactions" that are processed efficiently when the processTransaction() method is called.

By default, the router will process all actions as transactions. If transactionUse() is set to false, then all actions will get processed immediately, and cause immediate routing callbacks to all affected connectors after each action.

Parameters

in	*transactions*	A boolean value specifying whether to use transactions.

8.15.3.21 shouldContinueTransactionWithProgress()

```
bool Avoid::Router::shouldContinueTransactionWithProgress (
            unsigned int elapsedTime,
            unsigned int phaseNumber,
            unsigned int totalPhases,
            double proportion ) [virtual]
```

A method called at regular intervals during transaction processing to report progress and ask if the Router should continue the transaction.

You can subclass the Avoid::Router class to implement your own behaviour, such as to show a progress bar or cancel the transaction at the user's request.

Note that you can get a sense of progress by looking at the phaseNumber divided by the totalPhases and the progress in the current phase, but be aware that phases and the intervals and proportions at which this method is called will vary, sometime unpredictably.

You can return false to request that the Router abort the current transaction. Be aware that it may not abort in some phases. For others it may need to clean up some state before it is safe for you to interact with it again. Hence you should wait for a final call to this method with the phase Avoid::TransactionPhaseCompleted before continuing.

Note

Your implementation of this method should be very fast as it will be called many times. Also, you should not change or interact with the Router instance at all during these calls. Wait till you have received a call with the Avoid::TransactionPhaseCompleted phase.

Parameters

elapsedTime	The number of msec spent on the transaction since it began.
phaseNumber	A Router::TransactionPhases representing the current phase of the transaction.
totalPhases	The total number of phases to be performed during the transaction.
proportion	A double representing the progress in the current phase. Value will be between 0–1.

Returns

Whether the router should continue the transaction. This is true in the default (empty) implementation.

8.15.3.22 transactionUse()

```
bool Avoid::Router::transactionUse (
            void  ) const
```

Reports whether the router groups actions into transactions.

Returns

A boolean value describing whether transactions are in use.

See also

setTransactionUse
processTransaction

The documentation for this class was generated from the following files:

- router.h
- router.cpp

8.16 Avoid::ShapeConnectionPin Class Reference

The ShapeConnectionPin class represents a fixed point or "pin" on a shape that can be connected to.

```
#include <connectionpin.h>
```

Public Member Functions

- ShapeConnectionPin (ShapeRef ∗shape, const unsigned int classId, const double xOffset, const double y←Offset, const bool proportional, const double insideOffset, const ConnDirFlags visDirs)

  Constructs a ShapeConnectionPin at a specified absolute or proportional position relative to the parent shape.
- ShapeConnectionPin (JunctionRef ∗junction, const unsigned int classId, const ConnDirFlags visDirs=Conn←DirNone)

  Constructs a ShapeConnectionPin on a JunctionRef.
- void setConnectionCost (const double cost)

  Sets a cost used when selecting whether connectors should be be attached to this connection pin.
- const Point position (const Polygon &newPoly=Polygon()) const

  Returns the position of this connection pin.
- ConnDirFlags directions (void) const

  Returns the directions in which this connection pin has visibility.
- void setExclusive (const bool exclusive)

  Sets whether the pin is exclusive, i.e., only one connector can attach to it. This defaults to true for connection pins with visibility in a specific directions and false for pins with visibility in all directions.
- bool isExclusive (void) const

  Returns whether the connection pin is exclusive, i.e., only one connector can attach to it.

8.16.1 Detailed Description

The ShapeConnectionPin class represents a fixed point or "pin" on a shape that can be connected to.

A pin has a position that is specified relative to its parent shape.
When the shape is moved or resized, the pin will be automatically moved accordingly. Connectors attached to the pin will be rerouted.

Pins have a visibility direction and numeric ID used to identify them. This ID, known as their classId, may be shared by multiple pins on the same shape. You can use classIds when you want libavoid to choose from multiple potential choices (e.g., to specify multiple types of pins such as "input" or "output" pins on circuit elements).

If you would like connectors that attach to a single specific position on a shape, then just give each pin a unique classId (for that shape) and tell the connector to attach to that particular pin.

Pins may optionally be given a connection cost, via setConnectionCost(). In the case of multiple pins with the same classId, this causes the lower-cost pins to be chosen first, rather than libavoid choosing the best pin with that classId based solely on connector path cost.

Pins can be exclusive, which means subsequent connectors routed to the same classId will choose a different pin. Pins with a specified direction are exclusive by default, those with visibility in all directions are non-exclusive by default. This behaviour can be changed by calling the ShapeConnectionPin::setExclusive() method. Exclusive pins may only have a single connector attached to them.

8.16.2 Constructor & Destructor Documentation

8.16.2.1 ShapeConnectionPin() [1/2]

```
Avoid::ShapeConnectionPin::ShapeConnectionPin (
            ShapeRef * shape,
            const unsigned int classId,
            const double xOffset,
            const double yOffset,
            const bool proportional,
            const double insideOffset,
            const ConnDirFlags visDirs )
```

Constructs a ShapeConnectionPin at a specified absolute or proportional position relative to the parent shape.

Ownership of this ShapeConnectionPin is passed to the parent shape.

The connection point position offsets can be specified as absolute or proportional. If absolute, the xOffset and yOffset values are absolute offsets relative to the lower X and Y shape rectangle border positions. If proportional, the xOffset and yOffset values represent proportions of the shape's total width and height using a floating point value between 0 and 1.

Note that if you need the connection pin to appear at an exact position this may not be possible via proportional positions due to numerical inaccuracy in floating point multiplications. In this case you should use absolute offsets instead.

There are some predefined values for specifying the xOffset and yOffset arguments for proportional offsets:

- ATTACH_POS_TOP = 0
- ATTACH_POS_LEFT = 0
- ATTACH_POS_CENTRE = 0.5
- ATTACH_POS_BOTTOM = 1
- ATTACH_POS_RIGHT = 1 And two more for specifying absolute offsets:
- ATTACH_POS_MIN_OFFSET = offset of zero
- ATTACH_POS_MAX_OFFSET = offset of shape width/height

Importantly, shape connection pins will be moved automatically when the parent shape is moved or resized. Attachment for connectors will be chosen based on the classId specified to ConnEnd and these connectors will be subsequently rerouted.

If no value is given for the visDirs argument, then visibility is automatically determined based on the position of the connection point. Points on the shape boundary will have visibility from the shape out of that edge while points in the interior will have visibility in all directions. Note: Pins with visibility in a specific direction are exclusive by default, whereas those with visibility in all directions are non-exclusive by default.

The insideOffset argument can be used to set a distance to automatically offset the point within the shape. This is useful for orthogonal routing, where you usually want the connection point to lie inside the shape rather than exactly on its boundary. This offset will only be applied for connection pins specified with a position exactly on the shape boundary.

Parameters

in	*shape*	A pointer to the containing parent shape's ShapeRef.
in	*classId*	A non-zero integer used to identify this pin and other equivalent connection point, and used to specify attachment via the ConnEnd class.
in	*xOffset*	The X offset for the connection pin from the shape's lower X border position.
in	*yOffset*	The Y offset for the connection pin from the shape's lower Y border position.
in	*proportional*	A boolean specifying whether the X and Y offsets are proportional or not.
in	*insideOffset*	A distance to offset the connection point inside the shape if it lies on the boundary. Use 0.0 for no offset.
in	*visDirs*	One or more Avoid::ConnDirFlag options specifying the directions that this connection point has visibility. Use ConnDirNone to have visibility be directional if a pin is on the shape edge or in all directions otherwise.

8.16.2.2 ShapeConnectionPin() [2/2]

```
Avoid::ShapeConnectionPin::ShapeConnectionPin (
            JunctionRef * junction,
            const unsigned int classId,
            const ConnDirFlags visDirs = ConnDirNone )
```

Constructs a ShapeConnectionPin on a JunctionRef.

Ownership of this ShapeConnectionPin is passed to the parent junction.

This will usually be automatically called by the JunctionRef constructor to give the Junction four ShapeConnection↩ Pins, facing up, down, left and right.

Parameters

in	*junction*	A pointer to the containing parent junction's JunctionRef.
in	*classId*	An integer used to mark the class or group of this connection point, used for specifying attachment to ConnEnd.
in	*visDirs*	One or more Avoid::ConnDirFlag options specifying the directions that this connection point has visibility.

8.16.3 Member Function Documentation

8.16.3.1 directions()

```
ConnDirFlags Avoid::ShapeConnectionPin::directions (
             void ) const
```

Returns the directions in which this connection pin has visibility.

Returns

The visibility directions for this connection pin.

8.16.3.2 isExclusive()

```
bool Avoid::ShapeConnectionPin::isExclusive (
             void ) const
```

Returns whether the connection pin is exclusive, i.e., only one connector can attach to it.

Returns

A boolean denoting whether this pin is exclusive.

8.16.3.3 position()

```
const Point Avoid::ShapeConnectionPin::position (
             const Polygon & newPoly = Polygon() ) const
```

Returns the position of this connection pin.

Returns

The position of this connection pin.

8.16.3.4 setConnectionCost()

```
void Avoid::ShapeConnectionPin::setConnectionCost (
             const double cost )
```

Sets a cost used when selecting whether connectors should be be attached to this connection pin.

Parameters

in	*cost*	A routing cost applied to a route when selecting this connection pin.

8.16.3.5 setExclusive()

```
void Avoid::ShapeConnectionPin::setExclusive (
             const bool exclusive )
```

Sets whether the pin is exclusive, i.e., only one connector can attach to it. This defaults to true for connection pins with visibility in a specific directions and false for pins with visibility in all directions.

Parameters

in	*exclusive*	A bool representing whether this pin should be exclusive.

The documentation for this class was generated from the following files:

- connectionpin.h
- connectionpin.cpp

8.17 Avoid::ShapeRef Class Reference

The ShapeRef class represents a shape object.

```
#include <shape.h>
```

Inherits Avoid::Obstacle.

Public Member Functions

- ShapeRef (Router ∗router, Polygon &poly, const unsigned int id=0)

  Shape reference constructor.
- virtual ∼ShapeRef ()

  Shape reference destructor.
- const Polygon & polygon (void) const

  Returns a reference to the polygon boundary of this shape.
- void transformConnectionPinPositions (ShapeTransformationType transform)

  Adjusts all of the shape's connection pin positions and visibility directions for a given transformation type.
- unsigned int **id** (void) const

  Returns the ID of this obstacle.
- Router ∗ **router** (void) const

  Returns a pointer to the router scene this obstacle is in.

8.17.1 Detailed Description

The ShapeRef class represents a shape object.

Shapes are obstacles that connectors must be routed around. They can be placed into a Router scene and can be repositioned or resized (via Router::moveShape()).

Usually, it is expected that you would create a ShapeRef for each shape in your diagram and keep that reference in your own shape class.

8.17.2 Constructor & Destructor Documentation

8.17.2.1 ShapeRef()

```
Avoid::ShapeRef::ShapeRef (
            Router * router,
            Polygon & poly,
            const unsigned int id = 0 )
```

Shape reference constructor.

Creates a shape object reference, and adds it to the router scene. This shape will be considered to be an obstacle. This will cause connectors intersecting the newly added shape to be marked as needing to be rerouted.

If the router is using transactions, then changes will occur the next time Router::processTransaction() is called. See Router::setTransactionUse() for more information.

The shape can be moved with Router::moveShape() and removed from the scene and freed with Router::deleteShape().

The poly argument will usually be the boundary of the shape in your application with additional buffer of several pixels on each side. Specifying such a buffer results in connectors leaving a small amount of space around shapes, rather than touching them on the corners or edges.

Note

Regarding IDs: You can let libavoid manually handle IDs by not specifying them. Alternatively, you can specify all IDs yourself, but you must be careful to makes sure that each object in the scene (shape, connector, cluster, etc) is given a unique, positive ID. This uniqueness is checked if assertions are enabled, but if not and there are clashes then strange things can happen.

Parameters

in	*router*	The router scene to place the shape into.
in	*poly*	A Polygon representing the boundary of the shape.
in	*id*	Optionally, a positive integer ID unique among all objects.

8.17.2.2 ∼ShapeRef()

```
Avoid::ShapeRef::∼ShapeRef ( )  [virtual]
```

Shape reference destructor.

Do not call this yourself, instead call Router::deleteShape(). Ownership of this object belongs to the router scene.

8.17.3 Member Function Documentation

8.17.3.1 id()

```
unsigned int Avoid::Obstacle::id (
             void  ) const  [inherited]
```

Returns the ID of this obstacle.

Returns

The ID of the obstacle.

8.17.3.2 polygon()

```
const Polygon & Avoid::ShapeRef::polygon (
             void  ) const
```

Returns a reference to the polygon boundary of this shape.

Returns

A reference to the polygon boundary of the shape.

8.17.3.3 router()

```
Router * Avoid::Obstacle::router (
             void  ) const  [inherited]
```

Returns a pointer to the router scene this obstacle is in.

Returns

A pointer to the router scene for this obstacle.

8.17.3.4 transformConnectionPinPositions()

```
void Avoid::ShapeRef::transformConnectionPinPositions (
             ShapeTransformationType transform )
```

Adjusts all of the shape's connection pin positions and visibility directions for a given transformation type.

Parameters

in	*transform*	A ShapeTransformationType specifying the type of transform to be applied to all connection pins for the shape.

The documentation for this class was generated from the following files:

- shape.h
- shape.cpp

Chapter 9

File Documentation

9.1 connectionpin.h File Reference

Contains the interface for the ShapeConnectionPin class.

```
#include <cstdio>
#include <set>
#include <climits>
#include <utility>
#include "libavoid/dllexport.h"
#include "libavoid/connend.h"
#include "libavoid/geomtypes.h"
```

Classes

- class Avoid::ShapeConnectionPin

  The ShapeConnectionPin class represents a fixed point or "pin" on a shape that can be connected to.

Namespaces

- Avoid

  libavoid: Object-avoiding orthogonal and polyline connector routing library.

9.1.1 Detailed Description

Contains the interface for the ShapeConnectionPin class.

9.2 connector.h File Reference

Contains the interface for the ConnRef class.

```
#include <utility>
#include <list>
#include <vector>
#include "libavoid/dllexport.h"
#include "libavoid/vertices.h"
#include "libavoid/geometry.h"
#include "libavoid/connend.h"
```

Classes

- class Avoid::Checkpoint

  A checkpoint is a point that the route for a particular connector must visit. They may optionally be given an arrival/departure direction.
- class Avoid::ConnRef

  The ConnRef class represents a connector object.

Namespaces

- Avoid

  libavoid: Object-avoiding orthogonal and polyline connector routing library.

Typedefs

- typedef std::list< ConnRef ∗ > Avoid::ConnRefList

  A list of ConnRef objects.

Enumerations

- enum Avoid::ConnType { }

  Describes the type of routing that is performed for each connector.

9.2.1 Detailed Description

Contains the interface for the ConnRef class.

9.3 connend.h File Reference

Contains the interface for the ConnEnd class.

```
#include <cstdio>
#include <list>
#include <vector>
#include <utility>
#include "libavoid/dllexport.h"
#include "libavoid/geometry.h"
```

Classes

- class Avoid::ConnEnd

    The ConnEnd class represents different possible endpoints for connectors.

Namespaces

- Avoid

    libavoid: Object-avoiding orthogonal and polyline connector routing library.

Typedefs

- typedef unsigned int Avoid::ConnDirFlags

    One or more Avoid::ConnDirFlag options.

Enumerations

- enum Avoid::ConnDirFlag { }

    Flags that can be passed to the ConnEnd constructor to specify which sides of a shape this point should have visibility to if it is located within the shape's area.
- enum Avoid::ConnEndType { Avoid::ConnEndPoint , Avoid::ConnEndShapePin , Avoid::ConnEndJunction , Avoid::ConnEndEmpty }

    Types that describe the kind a connection that a ConnEnd represents.

9.3.1 Detailed Description

Contains the interface for the ConnEnd class.

9.4 geomtypes.h File Reference

Contains the interface for various geometry types and classes.

```
#include <cstdlib>
#include <vector>
#include <utility>
#include "libavoid/dllexport.h"
```

Classes

- class Avoid::Point

    The Point class defines a point in the plane.
- class Avoid::Box

    A bounding box, represented by the top-left and bottom-right corners.
- class Avoid::PolygonInterface

    A common interface used by the Polygon classes.
- class Avoid::Edge

    A line between two points.
- class Avoid::Polygon

    A dynamic Polygon, to which points can be easily added and removed.
- class Avoid::ReferencingPolygon

    A Polygon which just references its points from other Polygons.
- class Avoid::Rectangle

    A Rectangle, a simpler way to define the polygon for square or rectangular shapes.

Namespaces

- Avoid

  libavoid: Object-avoiding orthogonal and polyline connector routing library.

Typedefs

- typedef Point Avoid::Vector

  A vector, represented by the Point class.
- typedef Polygon Avoid::PolyLine

  A multi-segment line, represented with the Polygon class.

9.4.1 Detailed Description

Contains the interface for various geometry types and classes.

9.5 hyperedge.h File Reference

Contains the interface for the HyperedgeRerouter class.

```
#include <cstdio>
#include <list>
#include <vector>
#include <set>
#include "libavoid/dllexport.h"
```

Classes

- struct Avoid::HyperedgeNewAndDeletedObjectLists

  The HyperedgeNewAndDeletedObjectLists class stores lists of objects created and deleted during hyperedge improvement.
- class Avoid::HyperedgeRerouter

  The HyperedgeRerouter class is a convenience object that can be used to register hyperedges to be rerouted, improving the placement of their junctions and connector paths.

Namespaces

- Avoid

  libavoid: Object-avoiding orthogonal and polyline connector routing library.

Typedefs

- typedef std::list< ConnEnd > Avoid::ConnEndList

  A list of ConnEnd objects.
- typedef std::list< JunctionRef ∗ > Avoid::JunctionRefList

  A list of JunctionRef objects.

9.5.1 Detailed Description

Contains the interface for the HyperedgeRerouter class.

9.6 junction.h File Reference

Contains the interface for the JunctionRef class.

```
#include <list>
#include <set>
#include "libavoid/geomtypes.h"
#include "libavoid/obstacle.h"
#include "libavoid/dllexport.h"
```

Classes

- class Avoid::JunctionRef

    The JunctionRef class represents a fixed or free-floating point that connectors can be attached to.

Namespaces

- Avoid

    libavoid: Object-avoiding orthogonal and polyline connector routing library.

9.6.1 Detailed Description

Contains the interface for the JunctionRef class.

9.7 libavoid.h File Reference

Standard libavoid include file which includes all libavoid header files.

```
#include "libavoid/geomtypes.h"
#include "libavoid/shape.h"
#include "libavoid/connector.h"
#include "libavoid/connend.h"
#include "libavoid/graph.h"
#include "libavoid/debug.h"
#include "libavoid/timer.h"
#include "libavoid/vertices.h"
#include "libavoid/visibility.h"
#include "libavoid/router.h"
#include "libavoid/connectionpin.h"
#include "libavoid/junction.h"
#include "libavoid/viscluster.h"
```

Namespaces

- Avoid

  libavoid: Object-avoiding orthogonal and polyline connector routing library.

9.7.1 Detailed Description

Standard libavoid include file which includes all libavoid header files.

9.8 obstacle.h File Reference

Contains the interface for the Obstacle class, the superclass for ShapeRef and JunctionRef.

```
#include <list>
#include <set>
#include <cstdio>
#include "libavoid/geometry.h"
#include "libavoid/connectionpin.h"
```

Namespaces

- Avoid

  libavoid: Object-avoiding orthogonal and polyline connector routing library.

9.8.1 Detailed Description

Contains the interface for the Obstacle class, the superclass for ShapeRef and JunctionRef.

9.9 router.h File Reference

Contains the interface for the Router class.

```
#include <ctime>
#include <list>
#include <utility>
#include <string>
#include "libavoid/dllexport.h"
#include "libavoid/connector.h"
#include "libavoid/vertices.h"
#include "libavoid/graph.h"
#include "libavoid/timer.h"
#include "libavoid/hyperedge.h"
#include "libavoid/actioninfo.h"
#include "libavoid/hyperedgeimprover.h"
```

Classes

- class Avoid::Router

  The Router class represents a libavoid router instance.

Namespaces

- Avoid

  libavoid: Object-avoiding orthogonal and polyline connector routing library.

Enumerations

- enum Avoid::RouterFlag { Avoid::PolyLineRouting = 1 , Avoid::OrthogonalRouting = 2 }

  Flags that can be passed to the router during initialisation to specify options.
- enum Avoid::RoutingParameter {
  Avoid::segmentPenalty = 0 , Avoid::anglePenalty , Avoid::crossingPenalty , Avoid::clusterCrossingPenalty ,
  Avoid::fixedSharedPathPenalty , Avoid::portDirectionPenalty , Avoid::shapeBufferDistance , Avoid::idealNudgingDistance
  ,
  Avoid::reverseDirectionPenalty , **lastRoutingParameterMarker** }

  Types of routing parameters and penalties that can be used to tailor the style and improve the quality of the connector routes produced.
- enum Avoid::RoutingOption {
  Avoid::nudgeOrthogonalSegmentsConnectedToShapes = 0 , Avoid::improveHyperedgeRoutesMovingJunctions
  , Avoid::penaliseOrthogonalSharedPathsAtConnEnds , Avoid::nudgeOrthogonalTouchingColinearSegments ,
  Avoid::performUnifyingNudgingPreprocessingStep , Avoid::improveHyperedgeRoutesMovingAddingAndDeletingJunctions
  , Avoid::nudgeSharedPathsWithCommonEndPoint , **lastRoutingOptionMarker** }

  Types of routing options that can be enabled.
- enum Avoid::TransactionPhases {
  Avoid::TransactionPhaseOrthogonalVisibilityGraphScanX = 1 , Avoid::TransactionPhaseOrthogonalVisibilityGraphScanY
  , Avoid::TransactionPhaseRouteSearch , Avoid::TransactionPhaseCrossingDetection ,
  Avoid::TransactionPhaseRerouteSearch , Avoid::TransactionPhaseOrthogonalNudgingX , Avoid::TransactionPhaseOrthogonalN
  , Avoid::TransactionPhaseCompleted }

  Types of routing phases reported by Router::shouldContinueTransactionWithProgress().

9.9.1 Detailed Description

Contains the interface for the Router class.

9.10 shape.h File Reference

Contains the interface for the ShapeRef class.

```
#include <list>
#include <set>
#include <cstdio>
#include "libavoid/dllexport.h"
#include "libavoid/geometry.h"
#include "libavoid/obstacle.h"
```

Classes

- class Avoid::ShapeRef

  The ShapeRef class represents a shape object.

Namespaces

- Avoid

  libavoid: Object-avoiding orthogonal and polyline connector routing library.

Enumerations

- enum Avoid::ShapeTransformationType {
  Avoid::TransformationType_CW90 = 0 , Avoid::TransformationType_CW180 = 1 , Avoid::TransformationType_CW270 = 2 , Avoid::TransformationType_FlipX = 3 ,
  Avoid::TransformationType_FlipY = 4 }

  Describes the type of transformation that has been applied to a shape having its transformConnectionPinPositions() method called.

9.10.1 Detailed Description

Contains the interface for the ShapeRef class.

9.11 viscluster.h File Reference

Contains the interface for the ClusterRef class.

```
#include <list>
#include "libavoid/geometry.h"
#include "libavoid/dllexport.h"
```

Classes

- class Avoid::ClusterRef

  The ClusterRef class represents a cluster object.

Namespaces

- Avoid

  libavoid: Object-avoiding orthogonal and polyline connector routing library.

9.11.1 Detailed Description

Contains the interface for the ClusterRef class.

Index