The Clipper Library - Version 6

Library Overview Changes Example FAQ Rounding Deprecated License

Classes (Hierarchy)

- ClipperBase Clipper ClipperOffset PolyNode PolyTree

Types

CInt	InitOptions	JoinType	PolyFillType
ClipType	IntPoint	Path	PolyType
EndType	IntRect	Paths	ZFillCallback

Functions

Area	ClosedPathsFromPolyTree	OffsetPaths	PointInPolygon
CleanPolygon	MinkowskiDiff	OpenPathsFromPolyTree	PolyTreeToPaths
CleanPolygons	MinkowskiSum	Orientation	ReversePath

Units

ClipperLib

Copyright ©2010-2014 Angus Johnson - Clipper 6.2.0 - Help file built on 1-November-2014

Home	Overview	Clipper 🚸

Changes

v6.2.1 (31 October 2014)

Bugfix in **ClipperOffset.Execute** where the Polytree.IsHole property was returning incorrect values with negative offsets

Very minor improvement to join rounding in ClipperOffset

Fixed CPP OpenGL demo

v6.2.0 (26 September 2014)

Numerous minor bugfixes - too many to list. (See revisions 454-475 in Sourceforge Repository.)

Parameters have changed in the **custom callback function** that can be assigned to **ZFillFunction**.

Deprecated functions have been removed.

The Curves demo has been temporarily removed. (It needs to be updated but that was holding up this latest release of the Clipper library.) v6.1.3 (18 January 2014)

Fixed potential endless loop condition when adding open paths to Clipper.

Fixed missing implementation of **SimplifyPolygon** function in C++ code.

Fixed incorrect upper range constant for polygon coordinates in Delphi code.

Added:

Added **PointInPolygon** function.

Added:

Overloaded **MinkowskiSum** function to accommodate multi-contour paths.

Documentation updated.

v6.1.2 (15 December 2013)

Fixed broken C++ header file.

Minor improvement to joining polygons.

v6.1.1 (13 December 2013)

Fixed a couple of bugs affecting open paths that could raise unhandled exceptions.

v6.1.0 (2 December 2013)

Deleted:

Previously deprecated code has been removed.

The OffsetPaths function is now deprecated as it has been replaced by the **ClipperOffset** class which is much more flexible.

Several minor bugs have been fixed including occasionally an incorrect nesting within the PolyTree structure.

v6.0.0 (30 October 2013)

Ver 6 is a MAJOR update that adds support for open path clipping.

Added:

Open path (polyline) clipping. A new 'Curves' demo application showcases this (see the 'Curves' directory).

Update:

Major improvement in the merging of shared/collinear edges in clip solutions (see Execute).

Added:

The **IntPoint** structure now has an optional 'Z' member. (See the precompiler directive **use_xyz**.)
Users can now force Clipper to use 32bit integers (via the precompiler directive **use_int32**) instead of using 64bit integers. Using 32bit integers improves clipping performance by about 20-30%. (This has required the introduction of a new **cInt** integer type.) The trade-off to this speed up in performance is a very restricted vertex coordinate range of \pm 46340.

Modified:

To accommodate open paths, the **Polygon** and **Polygons** structures have been renamed **Path** and **Paths** respectively. The **AddPolygon** and **AddPolygons** methods of the ClipperBase class have been renamed **AddPath** and **AddPaths** respectively. Several other functions have been similarly renamed.

Modified:

The PolyNode Class has a new IsOpen property (to support open paths).

Modified:

The Clipper class has a new **ZFillFunction** property.

MinkowskiSum and MinkowskiDiff functions added.

Several other new functions have been added including **PolyTreeToPaths, OpenPathsFromPolyTree** and **ClosedPathsFromPolyTree**.

The Clipper constructor now accepts an optional InitOptions parameter to simplify setting properties.

Bugfix:

Numerous minor bugs have been fixed.

Deprecated:

Version 6 is a major upgrade from previous versions and **quite a number of changes have been made to exposed structures and functions**. To minimize inconvenience to existing library users, some code has been retained and some added to maintain backward compatibility. However, because this code will be removed in a future update, it has been marked as deprecated and a precompiler directive **use_deprecated** has been defined. v5.1.6 (23 May 2013)

BugFix:

The CleanPolygon function was buggy.

New OffsetPolyLines function. (Replaced with OffsetPaths function in ver 6.)

Changed:

The behaviour of the 'miter' JoinType has been changed so that when squaring occurs, it's no longer extended up to the miter limit but is squared off at exactly 'delta' units. (This improves the look of mitering with larger limits at acute angles.)

Update: Minor code refactoring and optimisations.

v5.1.5 (5 May 2013)

Update:

ForceSimple property added to the Clipper Class. (Renamed StrictlySimple in ver 6.)

Update:

Further improvements to this documentation.

Update: Minor code optimisations and trivial bug fixes.

v5.1.4 (24 March 2013)

Update: CleanPolygon function enhanced.

Update:

Documentation improved.

v5.1.3 (12 March 2013)

Bugfix:

Minor bug fixes.

Update:

Documentation significantly improved.

v5.1.2 (26 February 2013)

Bugfix:

PolyNode class was missing a constructor.

Update:

The *MiterLimit* parameter in the OffsetPolygons function has been renamed *Limit* and can now also be used to limit the number of vertices used to construct arcs when JoinType is set to jtRound.

v5.1.0 (17 February 2013)

Update:

ExPolygons has been replaced with the PolyTree / PolyNode classes to more fully represent the parent-child relationships of the polygons returned by Clipper's Execute method.

New CleanPolygon and CleanPolygons functions.

Bugfix:

Another orientation bug fixed.

v5.0.2 (30 December 2012)

Bugfix:

Significant fixes in and tidy of the internal Int128 class (which is used only when polygon coordinate values are greater than $\pm 0x3FFFFFFF$ (~1.07e9)).

Update: The Area function has been updated with a faster algorithm.

Update:

Documentation updates. The newish but undocumented 'CheckInputs' parameter of the OffsetPolygons function has been renamed 'AutoFix' and documented too. The comments on rounding have also been improved (ie clearer and expanded).

v4.10.0 (25 December 2012)
Orientation bugs should now be resolved (finally!).

Bug in Int128 class.

v4.9.8 (2 December 2012)

Further fixes to rare Orientation bug.

v4.9.7 (29 November 2012)

Bug that very rarely returned a wrong polygon orientation.

Obscure bug affecting OffsetPolygons when using jtRound for the JoinType parameter and when polygons also contain very large coordinate values.

v4.9.6 (9 November 2012)

Another obscure bug related to joining intersections.

v4.9.4 (2 November 2012)

Bugs in Int128 class occasionally causing wrong orientations.

Further fixes related to joining intersections.

v4.9.0 (9 October 2012)

Obscure bug related to joining intersections.

v4.8.9 (25 September 2012)

Obscure bug related to precision of intersections.

v4.8.8 (30 August 2012)

Fixed bug in OffsetPolygons function introduced in version 4.8.5.

v4.8.7 (24 August 2012)

ReversePolygon function in C++ translation was broken.

Two obscure bugs affecting orientation fixed too.

v4.8.6 (11 August 2012)

Potential for memory overflow errors when using ExPolygons structure.

Update:

ReversePolygons function was misnamed ReversePoints in C++.

Update: SimplifyPolygon function now takes a PolyFillType parameter.

v4.8.5 (15 July 2012)

Potential for memory overflow errors in OffsetPolygons().

v4.8.4 (1 June 2012)

Another obscure bug affecting ExPolygons structure.

v4.8.3 (27 May 2012)

Obscure bug causing incorrect removal of a vertex.

v4.8.2 (21 May 2012)

Obscure bug could cause an exception when using ExPolygon structure.

v4.8.1 (12 May 2012)

Update: Code tidy and minor bug fixes.
v4.8.0 (30 April 2012)

Bugfix:

Occasional errors in orientation fixed.

Update: Added notes on rounding to the documentation.

v4.7.6 (11 April 2012)

Bugfix:

Fixed a bug in the Orientation function (affecting C# translations only).

Update:

Minor documentation update.

v4.7.5 (28 March 2012)

Bugfix:

Fixed a recently introduced bug that occasionally caused an unhandled exception in C++ and C# translations.

v4.7.1 (3 March 2012)

Bugfix:

Rare crash when JoinCommonEdges joined polygons that 'cancelled' each other.

Bugfix:

Clipper's internal Orientation method occasionally returned wrong result.

Update:

Improved C# code (thanks to numerous suggestions from David Piepgrass)

v4.7 (10 February 2012)

Update:

Improved the joining of output polygons sharing a common edge.

v4.6.4 (8 December 2011)

Added:

Added SimplifyPolygon and SimplifyPolygons functions.

v4.6 (29 October 2011)

Added:

Support for **Positive** and **Negative** polygon fill types (in addition to the EvenOdd and NonZero fill types).

Bugfix:

The OffsetPolygons function was generating the occasional artefact when 'shrinking' polygons.

v4.5.5 (6 October 2011)

Bugfix:

Fixed an obscure bug in Clipper's JoinCommonEdges method.

Update:

Replaced IsClockwise function with Orientation function. The orientation issues affecting OffsetPolygons should finally be resolved.

Change:

The Area function once again returns the *signed* area.

v4.4.0 (6 August 2011)

BugFix:

A number of minor bugfixes that mostly affected the new ExPolygons structure.

v4.3.0 (17 June 2011)

Added:

ExPolygons structure that explicitly associates 'hole' polygons with their 'outer' container polygons.

Added:

Execute method now overloaded so the solution parameter can be either Polygons or ExPolygons.

Bug fix: Fixed a rare bug in the orientation of solution polygons.

v4.2.4 (26 April 2011)

Update:

Input polygon coordinates can now contain the full range of signed 64bit integers (ie +/-9,223,372,036,854,775,807). This means that floating point values can be converted to and from Clipper's 64bit vertex structure (IntPoint) and still retain a precision of up to 18 decimal places. However, since the large-integer math that supports this expanded range imposes a small cost on performance (~15%), a new property UseFullCoordinateRange has been added to the Clipper class to allow users the choice of whether or not to use this expanded coordinate range. If this property is disabled, coordinate values are restricted to +/-1,500,000,000. (By default, this property is enabled.)

v4.0 (4 April 2011)

Major update:

Clipper is a major rewrite of earlier versions. The biggest change is that floating point values are no longer used, except for the storing of edge slope values. The main benefit of this is the issue of numerical robustness has been addressed. Due to other major code improvements Clipper v4 is approximately 40% faster than Clipper v3.

Changes:

The AddPolyPolygon method has been renamed to AddPolygons, and the IgnoreOrientation property has been removed. The clipper_misc library has been merged back into the clipper library.

v2.8 (20 November 2010)

Updated:

Output polygons which previously shared a common edge are now merged.
Added:

Added support module for Cairo Graphics Library (with accompanying demo).

Added:

Added C# and C++ demos.

v2.522 (15 October 2010)

Added:

Added C# translation (thanks to Olivier Lejeune) and a link to Ruby bindings (thanks to Mike Owens).

v2.0 (31 July 2010)

Major update:

Clipper now processes polygons using both the Even-Odd (alternate) and Non-Zero (winding) filling rules.

v1.4c (16 June 2010)

Additions:

Added Delphi and C++ units to support the AGG graphics library

v1.2s (2 June 2010)

Additions:

C++ translation of Delphi code

v1.0 (9 May 2010) - Initial public release

See Also

Deprecated, Rounding, Clipper.Constructor, Clipper.Execute, Clipper.ZFillFunction, ClipperBase.AddPath, ClipperBase.AddPaths, ClipperOffset, ClipperOffset.Execute, PolyNode, PolyTree, Area, CleanPolygon, CleanPolygons, ClosedPathsFromPolyTree, MinkowskiDiff, MinkowskiSum, OffsetPaths, OpenPathsFromPolyTree, Orientation, PointInPolygon, PolyTreeToPaths, SimplifyPolygon, SimplifyPolygons, Defines, CInt, InitOptions, IntPoint, Path, Paths, PolyFillType, ZFillCallback

Home	Overview	Clipper 🚸

Deprecated

The precompiler directive '**use_deprecated**' allows users to update the Clipper library without being forced to make immediate changes to code that accesses the library. Depricated code will be removed in a future update. (Enabled by default.)

Deprecated types and functions:

All deprecated code has been removed from version 6.2.0.

Home	Overview	Clipper 🚸

Example

Delphi Code Sample:

```
uses
  graphics32, clipper;
function GetEllipsePoints(bounds: TIntRect): TPath;
begin
  //code to create an elliptical polygon here
end;
procedure DrawPolygons(polys: TPaths; color: TColor32);
begin
  //code to display the polygons here
end;
var
  sub, clp, sol: TPaths;
begin
  //set up the subject and clip polygons ...
  setlength(sub, 3);
  sub[0] := GetEllipsePoints(IntRect(100,100,300,300));
  sub[1] := GetEllipsePoints(IntRect(125,130,275,180));
  sub[2] := GetEllipsePoints(IntRect(125,220,275,270));
  setlength(clp, 1);
  clp[0] := GetEllipsePoints(IntRect(140,70,220,320));
  //display the subject and clip polygons ...
  DrawPolygons(sub, 0x8033FFFF);
  DrawPolygons(clp, 0x80FFFF33);
  //get the intersection of the subject and clip polygons ...
 with TClipper.Create do
  try
    AddPaths(sub, ptSubject, true);
    AddPaths(clp, ptClip, true);
    Execute(ctIntersection, sol, pftEvenOdd, pftEvenOdd);
  finally
    free;
  end;
  //finally draw the intersection polygons ...
  DrawPolygons(sol, 0x40808080);
```

C++ Code Sample:

```
#include "clipper.hpp"
```

//from clipper.hpp ...
//typedef long long cInt;
//struct IntPoint {cInt X; cInt Y;};
//typedef std::vector<IntPoint> Path;
//typedef std::vector<Polygon> Paths;

```
using namespace ClipperLib;
```

```
void GetEllipsePoints(IntRect& bounds, Path& p)
{/* ... */}
void DrawPolygons(Paths& p, unsigned color)
{/* ... */}
int main()
{
  //set up the subject and clip polygons ...
 Paths sub(3);
  GetEllipsePoints(IntRect(100,100,300,300), sub[0]);
  GetEllipsePoints(IntRect(125,130,275,180), sub[1]);
  GetEllipsePoints(IntRect(125,220,275,270), sub[2]);
  Paths clp(1);
  GetEllipsePoints(IntRect(140,70,220,320), clp[0]);
  //display the subject and clip polygons ...
  DrawPolygons(sub, 0x8033FFFF);
  DrawPolygons(clp, 0x80FFFF33);
  //get the intersection of the subject and clip polygons ...
  Clipper clpr;
  clpr.AddPaths(sub, ptSubject, true);
  clpr.AddPaths(clp, ptClip, true);
  Paths solution;
  clpr.Execute(ctIntersection, solution, pftEven0dd, pftEven0dd);
  //finally draw the intersection polygons ...
 DrawPolygons(solution, 0x40808080);
}
```

C# Code Sample:

```
using ClipperLib;
. . .
using Path = List<IntPoint>;
using Paths = List<List<IntPoint>>;
static Path GetEllipsePoints(IntRect bounds)
{/* ... */}
static void DrawPolygons(Path p, uint color)
\{/* \dots */\}
static void Main(string[] args)
{
  Paths subjs = new Paths(3);
  subjs.Add(GetEllipsePoints(new IntRect(100,100,300,300)));
  subjs.Add(GetEllipsePoints(new IntRect(125, 130, 275, 180)));
  subjs.Add(GetEllipsePoints(new IntRect(125,220,275,270)));
  Paths clips = new Paths(1);
  clips.Add(GetEllipsePoints(new IntRect(140,70,220,320)));
  DrawPolygons(subjs, 0x8033FFFF);
  DrawPolygons(clips, 0x80FFFF33);
  Paths solution = new Paths();
```

```
Clipper c = new Clipper();
c.AddPaths(subjs, PolyType.ptSubject, true);
c.AddPaths(clips, PolyType.ptClip, true);
c.Execute(ClipType.ctIntersection, solution);
DrawPolygons(solution, 0x40808080);
}
```



Copyright ©2010-2014 Angus Johnson - Clipper 6.2.0 - Help file built on 1-November-2014

Home	Overview	Clipper 🚸

FAQ

Why does Clipper use integer coordinates, not floats?

This has been done to preserve numerical robustness. Early versions of the library did use floating point coordinates, but it became apparent that floating point imprecision was always going to cause occasional errors.

How do I use floating point coordinates with Clipper?

It's a simple task to multiply your floating point coordinates by a scaling factor (that's typically a power of 10 depending on the desired precision). Then with the solution paths, divide the returned coordinates by this same scaling factor. Clipper accepts integer coordinates as large as $\pm 4.6e18$, so it can accommodate very large scaling.

Does Clipper handle polygons with holes?

'Holes' are defined by the specified polygon filling rule. (See also Clipper.Execute)

Some polygons in the solution share a common edge. Is this a bug?

No, though this should happen rarely as of version 6. (See Clipper.Execute for more about this.)

I have lots of polygons that I want to 'union'. Can I do this in one operation?

Yes. Just add all the polygons as subject polygons to the Clipper object. (You don't have to assign both subject and clip polygons.)

The polygons produced by ClipperOffset have tiny artefacts? Could this be a bug?

Make sure the input polygons don't self-intersect. Tiny self-intersections can sometimes be produced by previous clipping operations. These can be cleaned up using the CleanPolygon and CleanPolygons functions. Also, make sure the supplied polygons don't overlap. If they do, offset these separately. Finally, the precision of the input coordinates may be a problem. Because the Clipper Library only operates on integer coordinates, you may need to scale your coordinates (eg by a factor of 10) to improve precision.

Is there an easy way to reverse polygon orientations?

Yes, see **ReversePaths**.

My drawings contain lots of beziers, ellipses and arcs. How can I perform clipping operations on these?

You'll have to convert them to 'flattened' paths. For an example of how this can be done (and even reconstructed back into beziers, arcs etc), see the CurvesDemo application included in this library.

See Also

Clipper.Execute, ClipperOffset, CleanPolygon, CleanPolygons, ReversePaths, PolyFillType

Home	Overview	Clipper 🚸

License

The Clipper Library (including Delphi, C++ & C# source code, other accompanying code, examples and documentation), hereafter called "the Software", has been released under the following license, terms and conditions:

Boost Software License - Version 1.0 - August 17th, 2003 http://www.boost.org/LICENSE_1_0.txt

Permission is hereby granted, free of charge, to any person or organization obtaining a copy of the Software covered by this license to use, reproduce, display, distribute, execute, and transmit the Software, and to prepare derivative works of the Software, and to permit thirdparties to whom the Software is furnished to do so, all subject to the following:

The copyright notices in the Software and this entire statement, including the above license grant, this restriction and the following disclaimer, must be included in all copies of the Software, in whole or in part, and all derivative works of the Software, unless such copies or derivative works are solely in the form of machine-executable object code generated by a source language processor.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Home	Overview	Clipper 🚸

Rounding

By using an integer type for polygon coordinates, the Clipper Library has been able to avoid problems of **numerical robustness** that can cause havoc with geometric computations. Problems associated with integer **rounding** and their possible solutions are discussed below.



Nevertheless, the resulting imprecision can be very effectively managed by appropriate scaling.

Another complication of using a discrete numbers (as opposed to real numbers) is that very occasionally tiny **self-intersection** artefacts arise. In the unscaled image on the left (where one unit equals one pixel), the area of intersection of two polygons has been highlighted in bright green.

A 30X 'close up' of the lower points of intersection of these same two

polygons shows the presence of a tiny selfintersecting artefact. The three 'black dots' highlight the actual points of intersection (with their fractional coordinates displayed). The 'red dots' show where these points



of intersection are located once rounding is applied. With a little care you can see that rounding reverses the orientation of these vertices and causes a tiny self-intersecting artefact.

Although these tiny self-intersections are uncommon, if it's deemed necessary, they are best removed with CleanPolygons. (Setting Clipper's StrictlySimple property to true would also address this self-intersection but the tiny (sub-unit) polygon 'artefact' with incorrect orientation would still appear in the solution.)



In this final example, the single polygon on the left also has a tiny self-intersection. However, the clipping algorithm sees this vertex (88,50) as simply 'touching' rather than intersecting the right edge of the polygon (though only by a fraction of a unit). Since this intersection won't

normally be detected, the clipping solution (eg following a union operation) will still contain this tiny self-intersection. Setting Clipper's StrictlySimple property to true avoids this uncommon problem.

See Also

Clipper.StrictlySimple, CleanPolygons

🖆 Home	ClipperLib
--------	------------

4

CInt

```
Del.»
{$IFDEF use_int32}
 cInt = Int32;
{$ELSE}
  cInt = Int64;
{$ENDIF}
C++ »
#ifdef use_int32
  typedef int cInt;
#else
  typedef signed long long cInt;
#endif
C# »
#if use int32
  using cInt = Int32;
#else
  using cInt = Int64;
#endif
```

cInt is the integer type used by the Clipper Library to represent vertex coordinate values. (See also IntPoint.)

The library uses integers instead of floating point values to preserve numerical robustness. (Very early versions of the library used floating point coordinates, but it became apparent that floating point imprecision was always going to cause occasional errors.)

By default **cInt** represents a signed 64bit integer and polygon coordinates can have any value in the range \pm 9.2e+18. This accommodates the scaling of floating point coordinate values to very large integers so that very high degrees of precision can be retained during clipping. However, if coordinate values can be kept within the range \pm 3.0e+9, then by avoiding large integer math, a modest ~10% improvement in clipping performance is achieved.

If the preprocessor directive **use_int32** is defined, cInt will represent a

signed 32bit integer. This improves clipping performance by 20-30% but the trade-off is that coordinate values are restricted to the much narrower range of \pm 46340.

See Also

Defines, IntPoint

🖆 Home	ClipperLib
--------	------------

4

ClipType

```
Del.» type TClipType = (ctIntersection, ctUnion,
ctDifference, ctXor);
C++ » enum ClipType { ctIntersection, ctUnion,
ctDifference, ctXor };
C# » public enum ClipType { ctIntersection,
ctUnion, ctDifference, ctXor };
```

There are four boolean operations - AND, OR, NOT & XOR.

Given that subject and clip polygon brush 'filling' is defined both by their vertices and their respective filling rules, the four boolean operations can be applied to polygons to define new filling regions:

- AND (intersection) create regions where both subject **and** clip polygons are filled
- OR (union) create regions where **either** subject **or** clip polygons (or both) are filled
- NOT (difference) create regions where subject polygons are filled except where clip polygons are filled
- XOR (exclusive or) create regions where **either** subject **or** clip polygons are filled **but not** where **both** are filled

subject = blue & clip = orange





All polygon clipping is performed with a Clipper object with the specific boolean operation indicated by the *ClipType* parameter passed in its **Execute** method.

With regard to **open** paths (polylines), clipping rules generally match those of closed paths (polygons).

However, when there are both polyline and polygon subjects, the following clipping rules apply:

- union operations polylines will be clipped by any overlapping polygons so that non-overlapped portions will be returned in the solution together with the union-ed polygons
- intersection, difference and xor operations polylines will be clipped only by 'clip' polygons and there will be not interaction between polylines and subject polygons.

Example of clipping behaviour when mixing polyline and polygon subjects:


Overview, Clipper, Clipper.Execute, PolyFillType

🖆 Home	ClipperLib
--------	------------

EndType

```
Del.» type TEndType = (etClosedPolygon,
etClosedLine, etOpenSquare, etOpenRound,
etOpenButt);
C++ » enum EndType {etClosedPolygon,
etClosedLine, etOpenSquare, etOpenRound,
etOpenButt};
C# » public enum EndType {etClosedPolygon,
etClosedLine, etOpenSquare, etOpenRound,
etOpenButt};
```

The EndType enumerator has 5 values:

• **etClosedPolygon:** Ends are joined using the JoinType value and the path filled as a polygon

• **etClosedLine:** Ends are joined using the JoinType value and the path filled as a polyline

- **etOpenSquare:** Ends are *squared* off and extended **delta** units
- **etOpenRound:** Ends are *rounded* off and extended **delta** units
- **etOpenButt:** Ends are squared off with no extension.

• **etOpenSingle:** Offsets an open path in a single direction. Planned for a future update.

Note: With *etClosedPolygon* and *etClosedLine* types, the path closure will be the same regardless of whether or not the first and last vertices in the path match.





ClipperOffset.AddPath, ClipperOffset.AddPaths

🖆 Home	ClipperLib
--------	------------

InitOptions

```
Del.» type TInitOption = (ioReverseSolution,
ioStrictlySimple, ioPreserveCollinear);
C++ » enum InitOptions {
    ioReverseSolution = 1,
    ioStrictlySimple = 2,
    ioPreserveCollinear = 4};
C# » public const int ioReverseSolution = 1;
    public const int ioStrictlySimple = 2;
    public const int ioPreserveCollinear = 4;
```

Clipper.Constructor, Clipper.PreserveCollinear, Clipper.ReverseSolution, Clipper.StrictlySimple

🖆 Home	ClipperLib
--------	------------

IntPoint

```
Del.» TIntPoint = record X, Y: cInt; end;
C++ » struct IntPoint { cInt X; cInt Y; ... };
C# » public class IntPoint { public cInt X; {
get; set; } public cInt Y; { get; set; } ... };
```

The **IntPoint** structure is used to represent all vertices in the Clipper Library. An *integer* storage type has been deliberately chosen to preserve **numerical robustness**. (Early versions of the library used floating point coordinates, but it became apparent that floating point imprecision would always cause occasional errors.)

A sequence of IntPoints are contained within a Path structure to represent a single contour.

As of version 6, IntPoint now has an optional third member 'Z'. This can be enabled by exposing (ie uncommenting) the PreProcessor define 'use_xyz'. When the Z member is used, its values will be copied to corresponding verticies in solutions to clipping operations. However, at points of intersection where there's no corresponding Z value, the value will be assigned zero unless a new value is provided by a user supplied callback function.

Users wishing to clip or offset polygons containing *floating point* coordinates need to use appropriate scaling when converting these values to and from *IntPoints*.

See also the notes on rounding.

Rounding, Clipper.ZFillFunction, Defines, CInt, Path, Paths

🖆 Home	ClipperLib
--------	------------

IntRect

```
Del.» TIntRect = record left, top, right,
bottom: cInt; end;
C++ »
struct IntRect { cInt left; cInt top; cInt
right; cInt bottom; ... };
C# »
public class IntRect {
   public cInt left; { get; set; }
   public cInt top; { get; set; }
   public cInt right; { get; set; } ... };
```

Structure returned by Clipper's GetBounds method.

ClipperBase.GetBounds, CInt

🖆 Home	ClipperLib
--------	------------

JoinType

```
Del.» type TJoinType = (jtSquare, jtRound,
jtMiter);
C++ » enum JoinType {jtSquare, jtRound,
jtMiter};
C# » public enum JoinType {jtSquare, jtRound,
jtMiter};
```

When adding paths to a ClipperOffset object via the AddPaths method, the joinType parameter may be one of three types - jtMiter, jtSquare or jtRound.



• jtMiter: There's a necessary limit to mitered joins since offsetting edges that join at very acute angles will produce excessively long and narrow 'spikes'. To contain these potential spikes, the ClippOffset object's MiterLimit property specifies a maximum distance that vertices will be offset (*in multiples of delta*). For any given edge join, when miter offsetting would exceed that maximum distance, 'square' joining is applied.

jtRound: While flattened paths can never perfectly trace an arc, they are approximated by a series of arc chords (see ClipperObject's ArcTolerance property).

jtSquare: Squaring is applied uniformally at all convex edge joins at 1 × delta.

ClipperOffset, ClipperOffset.AddPaths, ClipperOffset.ArcTolerance, ClipperOffset.MiterLimit

🖆 Home	ClipperLib
--------	------------

Path

```
Del.» TPath = array of TIntPoint;
C++ » typedef std::vector<IntPoint> Path;
C# » using Path = List<IntPoint>;
```

This structure contains a sequence of IntPoint vertices defining a single contour (see also terminology). Paths may be *open* and represent a series of line segments bounded by 2 or more vertices, or they may be *closed* and represent polygons. Whether or not a path is open depends on context. *Closed* paths may be '*outer*' contours or '*hole*' contours. Which they are depends on orientation.

Multiple paths can be grouped into a Paths structure.

Overview, Example, ClipperBase.AddPath, PolyTree, Orientation, IntPoint, Paths

🖆 Home	ClipperLib
--------	------------

Paths

```
Del.» TPaths = array of TPath;
C++ » typedef std::vector< Path > Paths;
C# » using Paths = List<List< IntPoint >>;
```

This structure is fundamental to the Clipper Library. It's a list or array of one or more Path structures. (The *Path* structure contains an ordered list of vertices that make a single contour.)

Paths may *open* (a series of line segments), or they may *closed* (polygons). Whether or not a path is *open* depends on context. *Closed* paths may be '*outer*' contours or '*hole*' contours. Which they are depends on orientation.

Clipper.Execute, ClipperBase.AddPath, ClipperBase.AddPaths, OffsetPaths, IntPoint, Path

🖆 Home	ClipperLib
--------	------------

PolyFillType

```
Del.» type TPolyFillType = (pftEvenOdd,
pftNonZero, pftPositive, pftNegative);
C++ » enum PolyFillType {pftEvenOdd, pftNonZero,
pftPositive, pftNegative};
C# » public enum PolyFillType {pftEvenOdd,
pftNonZero, pftPositive, pftNegative};
```

Filling indicates those regions that are *inside* a closed path (ie 'filled' with a brush color or pattern in a graphical display) and those regions that are *outside*. The Clipper Library supports 4 filling rules: Even-Odd, Non-Zero, Positive and Negative.

The simplest filling rule is *Even-Odd* filling (sometimes called *alternate* filling). Given a group of closed paths start from a point outside the paths and progress along an imaginary line through the paths. When the first path is crossed the encountered region is filled. When the next path is crossed the encountered region is *not* filled. Likewise, each time a path is crossed, filling starts if it had stopped and stops if it had started.

With the exception of *Even-Odd* filling, all other filling rules rely on **edge direction** and **winding numbers** to determine filling. Edge direction is determined by the order in which vertices are declared when constructing a path. Edge direction is used to determine the **winding number** of each polygon subregion.

The winding number for each polygon sub-region can be derived by:

- 1. starting with a winding number of zero and
- 2. from a point (P1) that's outside all polygons, draw an imaginary line to a point that's inside a given sub-region (P2)
- 3. while traversing the line from P1 to P2, for each path that crosses the imaginary line from right to left increment the winding number, and for each path that crosses the line from left to right decrement the winding number.
- 4. Once you arrive at the given sub-region you have its winding number.



Even-Odd (Alternate): Odd numbered sub-regions are filled, while even numbered sub-regions are not.

Non-Zero (Winding): All non-zero sub-regions are filled. **Positive**: All sub-regions with winding counts > 0 are filled. **Negative**: All sub-regions with winding counts < 0 are filled.

Paths are added to a Clipper object using the AddPath or AddPaths methods and the filling rules (for subject and clip polygons separately) are specified in the Execute method.

Polygon regions are defined by one or more closed paths which may or may not intersect. A single polygon region can be defined by a single non-intersecting path or by multiple non-intersecting paths where there's typically an 'outer' path and one or more inner 'hole' paths. Looking at the three shapes in the image above, the middle shape consists of two concentric rectangles sharing the same clockwise orientation. With evenodd filling, where orientation can be disregarded, the inner rectangle would create a hole in the outer rectangular polygon. There would be no hole with non-zero filling. In the concentric rectangles on the right, where the inner rectangle is orientated opposite to the outer, a hole will be rendered with either even-odd or non-zero filling. A single path can also define multiple subregions if it self-intersects as in the example of the 5 pointed star shape below.



By far the most widely used fill rules are Even-Odd (aka Alternate) and Non-Zero (aka Winding). Most graphics rendering libraries (AGG, Android Graphics, Cairo, GDI+, OpenGL, Quartz 2D etc) and vector graphics storage formats (SVG, Postscript, Photoshop etc) support both these rules. However some libraries (eg Java's Graphics2D) only support one fill rule. *Android Graphics* and *OpenGL* are the only libraries (that I'm aware of) that support multiple filling rules.

It's useful to note that *edge direction* has no affect on a winding number's odd-ness or even-ness. (This is why orientation is ignored when the *Even-Odd* rule is employed.)

The direction of the Y-axis does affect polygon orientation and *edge direction*. However, changing Y-axis orientation will only change the *sign* of winding numbers, not their magnitudes, and has no effect on either *Even-Odd* or *Non-Zero* filling.

Clipper.Execute, ClipperBase.AddPath, ClipperBase.AddPaths, Orientation

🖆 Home	ClipperLib
--------	------------

PolyType

```
Del.» type TPolyType = (ptSubject, ptClip);
C++ » enum PolyType { ptSubject, ptClip };
C# » public enum PolyType { ptSubject, ptClip
};
```

Boolean (clipping) operations are mostly applied to two sets of Polygons, represented in this library as *subject* and *clip* polygons. Whenever Polygons are added to the Clipper object, they must be assigned to either subject or clip polygons.

UNION operations can be performed on one set or both sets of polygons, but all other boolean operations require both sets of polygons to derive meaningful solutions.

ClipperBase.AddPath, ClipperBase.AddPaths, ClipType

🖆 Home	ClipperLib
--------	------------

ZFillCallback

Del.» type TZFillCallback = procedure (const E1Bot, E1Top, E2Bot, E2Top: TIntPoint; var Pt: TIntPoint);

C++ » typedef void (*ZFillCallback)(const IntPoint& e1bot, IntPoint& e1top, IntPoint& e2bot, IntPoint& e2top, IntPoint& pt);

C# >> public delegate void ZFillCallback(IntPoint bot1, IntPoint top1, IntPoint bot2, IntPoint top2, ref IntPoint pt);

If the **use_xyz** pre-processor directive is enabled, then the IntPoint class will have an extra 'Z' member and the Clipper class's ZFillFunction property will be exposed so it can be assigned a custom callback function.

This custom callback procedure requires five IntPoint parameters: the first 2 parameters are the vertices that define one line segment involved in the intersection and the next two parameters the other line segment. (Since the Clipper library has been developed in an environment that uses an inverted Y axis display, e1bot and e2bot will always have Y values greater than or equal to their corresponding e1top and e2top Y values.) The last IntPoint parameter contain the actual coordinates at the intersection. This last parameter is passed by reference so that its Z member can be assigned with a custom value.

Clipper.ZFillFunction, Defines

🖆 Home	ClipperLib
--------	------------

Area

```
Del.» function Area(const pts: TPath): double;
C++ » double Area(const Path &poly);
C# » public static double Area(Path poly);
```

This function returns the area of the supplied polygon. It's assumed that the path is closed and does not self-intersect. Depending on orientation, this value may be positive or negative. If Orientation is true, then the area will be positive and conversely, if Orientation is false, then the area will be negative.
Orientation, Path

🖆 Home	ClipperLib
--------	------------

CleanPolygon

```
Del.» function CleanPolygon(const Poly: TPath;
Distance: double = 1.415): TPath;
C++ » void CleanPolygon(const Path &in_poly,
Path &out_poly, double distance = 1.415);
C++ » void CleanPolygon(Path &poly, double
distance = 1.415);
C# » public static Path CleanPolygon(Path poly,
double distance = 1.415);
```

Removes vertices:

- that join co-linear edges, or join edges that are almost co-linear (such that if the vertex was moved no more than the specified distance the edges would be co-linear)
- that are within the specified distance of an adjacent vertex
- that are within the specified distance of a semi-adjacent vertex together with their out-lying vertices

Vertices are *semi-adjacent* when they are separated by a single (out-lying) vertex.

The *distance* parameter's default value is approximately $\sqrt{2}$ so that a vertex will be removed when adjacent or semi-adjacent vertices having their corresponding X and Y coordinates differing by no more than 1 unit. (If the egdes are semi-adjacent the out-lying vertex will be removed too.)

C++ only: This function is overloaded. In the first definition, the *in_poly* and *out_poly* parameters can reference the same *Path* object though in that case the calling code might be clearer if the second definition (accepting a single *Paths* parameter) is used.





pts = [(10,10), (100,10), (100,100), (10,100), (99,99)]

pts = [(10,10), (100,10), (100,100), (10,100), (10,50)]

CleanPolygons, SimplifyPolygon, Path

🖆 Home	ClipperLib
--------	------------

CleanPolygons

```
Del.» function CleanPolygons(const Polys:
TPaths; Distance: double = 1.415): TPaths;
C++ » void CleanPolygons(const Paths &in_polys,
Paths &out_polys, double distance = 1.415);
C++ » void CleanPolygons(Paths &polys, double
distance = 1.415);
C# » public static Paths CleanPolygons(Paths
polys, double distance = 1.415);
```

Removes vertices:

- that join co-linear edges, or join edges that are almost co-linear (such that if the vertex was moved no more than the specified distance the edges would be co-linear)
- that are within the specified distance of an adjacent vertex
- that are within the specified distance of a semi-adjacent vertex together with their out-lying vertices

Vertices are *semi-adjacent* when they are separated by a single (out-lying) vertex.

The *distance* parameter's default value is approximately $\sqrt{2}$ so that a vertex will be removed when adjacent or semi-adjacent vertices having their corresponding X and Y coordinates differing by no more than 1 unit. (If the egdes are semi-adjacent the out-lying vertex will be removed too.)

C++ only: This function is overloaded. In the first definition, the *in_polys* and *out_polys* parameters can reference the same *Paths* object though in that case the calling code might be clearer if the second definition (accepting a single *Paths* parameter) is used.





pts = [(10,10), (100,10), (100,100), (10,100), (99,99)]

pts = [(10,10), (100,10), (100,100), (10,100), (10,50)]

CleanPolygon, SimplifyPolygons

🖆 Home	ClipperLib
--------	------------

ClosedPathsFromPolyTree

```
Del.» function ClosedPathsFromPolyTree(PolyTree:
TPolyTree): TPaths;
C++ » void ClosedPathsFromPolyTree(PolyTree&
polytree, Paths& paths);
C# » public static void
ClosedPathsFromPolyTree(PolyTree polytree, Paths
paths);
```

This function filters out *open* paths from the **PolyTree** structure and returns only *closed* paths in a **Paths** structure.

PolyTree, Paths

🖆 Home	ClipperLib
--------	------------

MinkowskiDiff

```
Del.» function MinkowskiDiff(const Poly1: TPath;
const Poly2: TPath): TPaths;
C++ » void MinkowskiDiff(const Path& poly1,
const Path& poly2, Paths& solution);
C# » public static Paths MinkowskiDiff(Path
poly1, Path poly2);
```

Minkowski Difference is performed by *subtracting* each point in a polygon from the set of points in an open or closed path. A key feature of Minkowski Difference is that when it's applied to two polygons, the resulting polygon will contain the coordinate space origin whenever the two polygons touch or overlap. (This function is often used to determine when polygons collide.)

In the image on the left the blue polygon is the 'minkowski difference' of the two red boxes. The black dot represents the coordinate space origin.

MinkowskiSum, Path

🖆 Home	ClipperLib
--------	------------

MinkowskiSum

```
Del.» function MinkowskiSum(const Pattern:
TPath; const Path: TPath; PathIsClosed:
Boolean): TPaths; overload;
Del.» function MinkowskiSum(const Pattern:
TPath; const Paths: TPaths; PathFillType:
TPolyFillType; PathIsClosed: Boolean): TPaths;
overload;
C++ » void MinkowskiSum(const Path& pattern,
const Path& path, Paths& solution, bool
pathIsClosed);
C++ » void MinkowskiSum(const Path& pattern,
const Paths& paths, Paths& solution,
PolyFillType pathFillType, bool pathIsClosed);
   » public static Paths MinkowskiSum(Path
C#
pattern, Path path, bool pathIsClosed);
    » public static Paths MinkowskiSum(Path
C#
pattern, Paths paths, PolyFillType pathFillType,
```

```
bool pathIsClosed);
```

Minkowski Addition is performed by *adding* each point in a polygon 'pattern' to the set of points in an open or closed path. The resulting polygon (or polygons) defines the region that the 'pattern' would pass over in moving from the beginning to the end of the 'path'.

```
Path path = new Path();
Path pattern = new Path();
Paths solution = new Paths();
//Greek capital sigma (sum sign) ...
Int64[] ints1 = new Int64[] { 300, 400, 100, 400, 200, 300, 10
path = IntsToPolygon(ints1);
//diagonal brush pattern ...
Int64[] ints2 = new Int64[] { 4, -6, 6, -6, -4, 6, -6, 6 };
pattern = IntsToPolygon(ints2);
solution = Clipper.MinkowskiSum(pattern, path, false);
//move 'pattern' to the end of 'path' ...
```

```
pattern = TranslatePath(pattern, 300, 200);
//Display solution ± pattern ...
```

MinkowskiDiff

🖆 Home	ClipperLib
--------	------------

OffsetPaths

```
Del.» function OffsetPaths(const polys: Paths;
const delta: double; JoinType: TJoinType =
jtSquare; EndType: TEndType = etClosed; Limit:
double = 0.0): TPaths;
```

```
C++ » void OffsetPaths(const Paths &in_polys,
Paths &out_polys, double delta, JoinType
jointype = jtSquare, EndType endtype = etClosed,
double limit = 0.0);
```

```
C# >> public static Paths OffsetPaths(Paths
polys, double delta, JoinType jointype =
JoinType.jtSquare, EndType endtype =
EndType.etClosed, double limit = 0.0);
```

Deprecated. (See ClipperOffset.)

ClipperOffset

🖆 Home	ClipperLib
--------	------------

OpenPathsFromPolyTree

```
Del.» function OpenPathsFromPolyTree(PolyTree:
TPolyTree): TPaths;
C++ » void OpenPathsFromPolyTree(PolyTree&
polytree, Paths& paths);
C# » public static void
OpenPathsFromPolyTree(PolyTree polytree, Paths
paths);
```

This function filters out *closed* paths from the **PolyTree** structure and returns only *open* paths in a **Paths** structure.

PolyTree, Paths

🖆 Home	ClipperLib
--------	------------

Orientation

Del.» function Orientation(const poly: TPath):
boolean;

C++ » **bool** Orientation(**const** Path &poly); // Function in the ClipperLib namespace.

C# » public static bool Orientation(Path poly);
// Static method of the Clipper class in the
ClipperLib namespace.

Orientation is only important to *closed* paths. Given that vertices are declared in a specific order, orientation refers to the direction (clockwise or counter-clockwise) that these vertices progress around a closed path.

Orientation is also dependent on axis direction:

• On **Y-axis positive** *upward* **displays**, Orientation will return true if the polygon's orientation is counter-clockwise.

• On **Y-axis positive** *downward* **displays**, Orientation will return true if the polygon's orientation is clockwise.



Notes:

• Self-intersecting polygons have indeterminate orientations in which case this function won't return a meaningful value.

• The majority of 2D graphic display libraries (eg GDI, GDI+, XLib, Cairo, AGG, Graphics32) and even the SVG file format have their coordinate origins at the top-left corner of their respective viewports *with their Y axes increasing downward*. However, some display

libraries (eg Quartz, OpenGL) have their coordinate origins undefined or in the classic bottom-left position with their Y axes increasing upward.

• For Non-Zero filled polygons, the orientation of holes *must be opposite* that of outer polygons.

• For closed paths (polygons) in the *solution* returned by Clipper's Execute method, their orientations will always be *true* for outer polygons and *false* for hole polygons (unless the **ReverseSolution** property has been enabled).

Overview, Clipper.ReverseSolution, Path

🖆 Home	ClipperLib
--------	------------

PointInPolygon

Del.» function PointInPolygon(const Pt: TIntPoint; const poly: TPath): Integer; C++ » int PointInPolygon(const IntPoint pt, const Path &poly); // Function in the ClipperLib namespace. C# » public static int PointInPolygon(IntPoint pt, Path poly); // Static method of the Clipper class.

Returns 0 when false, -1 when pt is **on** poly and +1 when pt is **in** poly.

It's assumed that 'poly' is closed and does not self-intersect.

IntPoint, Path

🖆 Home	ClipperLib
--------	------------

PolyTreeToPaths

Del.» function PolyTreeToPaths(PolyTree: TPolyTree): TPaths;

C++ » void PolyTreeToPaths(PolyTree& polytree, Paths& paths);

C# >> public static Paths
PolyTreeToPaths(PolyTree polytree);

This function converts a **PolyTree** structure into a **Paths** structure.

PolyTree, Paths

🖆 Home	ClipperLib
--------	------------
ReversePath

Del.» function ReversePath(const polys: TPath):
TPath;
C++ » void ReversePath(const Path &p);
C# » //Call Path.Reverse().

Reverses the vertex order (and hence orientation) in the specified path.

Path

🖆 Home	ClipperLib
--------	------------

4

ReversePaths

```
Del.» function ReversePaths(const p: TPaths):
TPaths;
C++ » void ReversePaths(const Paths &p);
C# » void ReversePaths( Paths p );
```

Reverses the vertex order (and hence orientation) in each contained path.

🖆 Home	ClipperLib
--------	------------

4

SimplifyPolygon

```
Del.» function SimplifyPolygon(const Poly:
TPath; FillType: TPolyFillType = pftEvenOdd):
TPaths;
C++ » void SimplifyPolygon(const Path &in_poly,
Paths &out_polys,
PolyFillType fillType = pftEvenOdd);
C# » public static Paths SimplifyPolygon(Path
poly,
PolyFillType fillType =
PolyFillType.pftEvenOdd);
```

Removes self-intersections from the supplied polygon (by performing a boolean *union* operation using the nominated PolyFillType). Polygons with non-contiguous duplicate vertices (ie 'touching') will be split into two polygons.

Note: There's currently no guarantee that polygons will be strictly simple since 'simplifying' is still a work in progress.







Clipper.StrictlySimple, CleanPolygon, Path, PolyFillType

🖆 Home	ClipperLib
--------	------------

4

SimplifyPolygons

```
Del.» function SimplifyPolygons(const polys:
TPaths;
        FillType: TPolyFillType = pftEvenOdd):
TPaths;
C++ » void SimplifyPolygons(const Paths
&in_polys, Paths &out_polys,
        PolyFillType fillType = pftEvenOdd);
C++ » void SimplifyPolygons(Paths &polys,
PolyFillType fillType = pftEvenOdd);
C# » public static Polygons
SimplifyPolygons(Paths polys,
        PolyFillType fillType =
PolyFillType.pftEvenOdd);
```

Removes self-intersections from the supplied polygons (by performing a boolean *union* operation using the nominated PolyFillType). Polygons with non-contiguous duplicate vertices (ie 'vertices are touching') will be split into two polygons.

C++ only: This function is overloaded. In the first definition, the *in_polys* and *out_polys* parameters can reference the same *Paths* object though in that case the calling code might be clearer if the second definition (accepting a single *Paths* parameter) is used.

Note: There's currently no guarantee that polygons will be strictly simple since 'simplifying' is still a work in progress.







Clipper.StrictlySimple, CleanPolygons, PolyFillType

☆Home &Clippe	rLib ^k Clipper	Clipper 🎡
---------------	---------------------------	-----------

Clipper.Constructor

```
Del.» constructor TClipper.Create(InitOptions:
TInitOptions = []);
C++ » Clipper::Clipper(int initOptions = 0) :
ClipperBase();
C# » public Clipper(initOptions = 0): base()
{};
```

The Clipper constructor creates an instance of the Clipper class. One or more InitOptions may be passed as a parameter to set the corresponding properties. (These properties can still be set or reset after construction.)

Examples:

```
//C++ constructor setting StrictlySimple and PreserveCollinear properties ...
Clipper clipper(ioStrictlySimple | ioPreserveCollinear);
//C# constructor setting StrictlySimple and PreserveCollinear properties ...
Clipper clipper = new Clipper(Clipper.ioStrictlySimple | Clipper.ioPreserveColline)
```

//Delphi constructor setting StrictlySimple and PreserveCollinear properties ...
clipper := TClipper.Create([ioStrictlySimple, ioPreserveCollinear]);

PreserveCollinear, ReverseSolution, StrictlySimple, InitOptions

☆Home &Clippe	rLib ^k Clipper	Clipper 🎡
---------------	---------------------------	-----------

Clipper.Execute

```
Del.» function Execute(clipType: TClipType;
  out solution: TPaths;
  subjFillType: TPolyFillType = pftEvenOdd;
  clipFillType: TPolyFillType = pftEvenOdd):
boolean; overload;
function Execute(clipType: TClipType;
  out solution: TPolyTree;
  subjFillType: TPolyFillType = pftEvenOdd;
  clipFillType: TPolyFillType = pftEvenOdd):
boolean; overload;
C^{++} \gg
bool Execute(ClipType clipType,
  Paths & solution,
  PolyFillType subjFillType = pftEvenOdd,
 PolyFillType clipFillType = pftEvenOdd);
bool Execute(ClipType clipType,
  PolyTree &solution,
 PolyFillType subjFillType = pftEvenOdd,
 PolyFillType clipFillType = pftEvenOdd);
C# »
public bool Execute(ClipType clipType,
  Paths solution,
  PolyFillType subjFillType,
  PolyFillType clipFillType);
public bool Execute(ClipType clipType,
  PolyTree solution,
 PolyFillType subjFillType,
 PolyFillType clipFillType);
```

Once *subject* and *clip* paths have been assigned (via AddPath and/or AddPaths), **Execute** can then perform the clipping operation (intersection, union, difference or XOR) specified by the clipType

parameter.

The **solution** parameter can be either a Paths or PolyTree structure. The Paths structure is simpler than the PolyTree stucture. Because of this it is quicker to populate and hence clipping performance is a little better (it's roughly 10% faster). However, the PolyTree data structure provides more information about the returned paths which may be important to users. Firstly, the PolyTree structure preserves nested parent-child polygon relationships (ie outer polygons owning/containing *holes* and holes owning/containing other outer polygons etc). Also, **only the PolyTree structure can differentiate between open and closed paths** since each PolyNode has an **IsOpen** property. (The **Path** structure has no member indicating whether it's open or closed.) For this reason, **when open paths are passed to a Clipper object, the user must use a PolyTree object as the solution parameter**, otherwise an exception will be raised.

When a PolyTree object is used in a clipping operation on **open** paths, two ancilliary functions have been provided to quickly separate out *open* and *closed* paths from the solution - **OpenPathsFromPolyTree** and **ClosedPathsFromPolyTree**. **PolyTreeToPaths** is also available to convert path data to a Paths structure (irrespective of whether they're *open* or *closed*).

There are several things to note about the solution paths returned:

they aren't in any



A single union operation on many square polygons showing by the absence of 'keyholes', Clipper's ability to merge polygons with common edges.

specific order

they should never overlap or be self-intersecting (but see notes on rounding)

- holes will be oriented opposite outer polygons
- the **solution fill type** can be considered either *EvenOdd* or *NonZero* since it will comply with either filling rule
- polygons may rarely share a common edge (though this is now very rare as of version 6)

The **subjFillType** and **clipFillType** parameters define the polygon **fill rule** to be applied to the polygons (ie closed paths) in the subject and clip paths respectively. (It's usual though obviously not essential that both sets of polygons use the same fill rule.)

Execute can be called multiple times without reassigning subject and clip polygons (ie when different clipping operations are required on the same polygon sets).

Example, Rounding, ClipperBase.AddPath, ClipperBase.AddPaths, PolyNode.IsOpen, PolyTree, ClosedPathsFromPolyTree, OpenPathsFromPolyTree, PolyTreeToPaths, ClipType, Path, Paths, PolyFillType

☆Home &Clippe	rLib ^k Clipper	Clipper 🎡
---------------	---------------------------	-----------

Clipper.PreserveCollinear

```
Del.» property PreserveCollinear: boolean;
override;
C++ » void PreserveCollinear(bool value);
C# » public bool PreserveCollinear { get {} set
{} };
```

By default, when three or more vertices are collinear in input polygons (subject or clip), the Clipper object removes the 'inner' vertices before clipping. When enabled the PreserveCollinear property prevents this default behavior to allow these inner vertices to appear in the solution.

Constructor

☆Home &Clippe	rLib ^k Clipper	Clipper 🎡
---------------	---------------------------	-----------

Clipper.ReverseSolution

```
Del.» property ReverseSolution: boolean;
override;
C++ » void ReverseSolution(bool value);
C# » public bool ReverseSolution { get {} set
{} };
```

When this property is set to true, polygons returned in the **solution** parameter of the **Execute()** method will have orientations opposite to their normal orientations.

Execute, Orientation

☆Home &Clippe	rLib ^k Clipper	Clipper 🎡
---------------	---------------------------	-----------

Clipper.StrictlySimple

```
Del.» property StrictlySimple: boolean;
override;
C++ » void StrictlySimple(bool value);
C# » public bool StrictlySimple { get {} set {}
};
```

Terminology:

- A *simple* polygon is one that does not self-intersect.
- A *weakly simple* polygon is a simple polygon that contains 'touching' vertices, or 'touching' edges.
- A *strictly simple* polygon is a simple polygon that does not contain 'touching' vertices, or 'touching' edges.

Vertices 'touch' if they share the same coordinates (and are not adjacent). An edge touches another if one of its end vertices touches another edge excluding its adjacent edges, or if they are co-linear and overlapping (including adjacent edges).

Polygons returned by clipping operations (see Clipper.Execute()) should always be simple polygons. When the **StrictlySimply** property is enabled, polygons returned will be *strictly simple*, otherwise they may be *weakly simple*. It's **computationally expensive** ensuring polygons are *strictly simple* and so this property is disabled by default.

Note: There's currently no guarantee that polygons will be strictly simple since 'simplifying' is still a work in progress.



In the image above, the two examples show weakly simple polygons being broken into two strictly simple polygons. (The outlines with arrows are intended to aid visualizing vertex order.)

See also the article on Simple Polygons on Wikipedia.

Execute, SimplifyPolygons

☆Home &Clippe	rLib ^k Clipper	Clipper 🎡
---------------	---------------------------	-----------

Clipper.ZFillFunction

Del.» property ZFillFunction: TZFillCallback
read FZFillCallback write FZFillCallback;
C++ » void ZFillFunction(ZFillCallback
zFillFunc);
C# » public ZFillCallback ZFillFunction { get;
set; };

This property is only exposed when the pre-processor directive use_xyz has been defined. If this is the case, a 'Z' member will be included in the IntPoint structure where users can store custom data. While most vertices in a clipping solution will correspond to input (subject and clip) vertices, there will also be new vertices wherever edges intersect. This property assigns a custom callback function to the Clipper object so that custom 'Z' values can be assigned to these intersection vertices. (Note that 'Z' values in the solution at non-intersecting vertices will simply be copied from matching input vertices along with the X and Y values.)

It is up to the library user to assign 'Z' values for new intersection vertices (otherwise these values will remain 0). The four vertices that define the intersecting line segments will be passed to the callback function (together with the new intersection vertex) to aid the user in determining appropriate Z values.

Defines, IntPoint, ZFillCallback

☆Home &ClipperLib *ClipperBase Clipper
--

ClipperBase.AddPath

Del.» function AddPath(const path: TPath;
polyType: TPolyType; Closed: Boolean): Boolean;
C++ » bool AddPath(const Path &pg, PolyType

polyType, **bool** closed);

C# >> public virtual bool AddPath(Path pg, PolyType polyType, bool closed);

Any number of subject and clip paths can be added to a clipping task, either individually via the AddPath() method, or as groups via the AddPaths() method, or even using both methods.



'Subject' paths may be either *open* (lines) or *closed* (polygons) or even

a mixture of both, but 'clipping' paths must always be *closed*. Clipper allows polygons to clip both lines and other polygons, but doesn't allow lines to clip either lines or polygons.

With closed paths, orientation should conform with the filling rule that will be passed via Clippper's Execute method.

Path Coordinate range:

Return Value:

The function will return false if the path is invalid for clipping. A path is

invalid for clipping when:

- it has less than 2 vertices
- it has 2 vertices but is not an open path
- the vertices are all co-linear and it is not an open path
Example, Clipper.Execute, AddPaths, Orientation, Defines, Path, PolyFillType, PolyType

☆Home &ClipperLib *ClipperBase Clipper
--

ClipperBase.AddPaths

Del.» **function** AddPaths(**const** paths: **TPaths**; polyType: **TPolyType**; Closed: Boolean): boolean;

C++ >> bool AddPaths(const Paths &ppg, PolyType
polyType, bool closed);

C# >> public virtual bool AddPaths(Paths ppg, PolyType polyType, bool closed);

Any number of subject and clip paths can be added to a clipping task, either individually via the AddPath() method, or as groups via the AddPaths() method, or even using both methods.

'Subject' paths may be either *open* (lines) or *closed* (polygons) or even

a mixture of both, but 'clipping' paths must always be *closed*. Clipper allows polygons to clip both lines and other polygons, but doesn't allow lines to clip either lines or polygons.

With closed paths, orientation should conform with the filling rule that will be passed via Clippper's Execute method.

Path Coordinate range:

Return Value:

The function will return false if the path is invalid for clipping. A path is

invalid for clipping when:

- it has less than 2 vertices
- it has 2 vertices but is not an open path
- the vertices are all co-linear and it is not an open path

Example, Clipper.Execute, AddPaths, Orientation, Defines, Paths, PolyFillType, PolyType

☆Home &ClipperLib *ClipperBase Clipper
--

ClipperBase.Clear

```
Del.» procedure Clear;
C++ » virtual void Clear();
C# » public void Clear() {};
```

The **Clear** method removes any existing subject and clip polygons allowing the Clipper object to be reused for clipping operations on *different* polygon sets.

☆Home &ClipperLib *ClipperBase Clipper
--

ClipperBase.GetBounds

Del.» function GetBounds: TIntRect; C++ » IntRect GetBounds(); C# » public IntRect GetBounds() {...};

This method returns the axis-aligned bounding rectangle of all polygons that have been added to the Clipper object.

Example, IntRect

∆ Home	ClipperLib	ClipperOffset	Clipper 🏡
---------------	------------	---------------	-----------

ClipperOffset.AddPath

Del.» procedure AddPath(const Path: TPath; JoinType: TJoinType; EndType: TEndType); C++ » void AddPath(const Path& path, JoinType jointype, EndType endtype); C# » public void AddPath(Path path, JoinType jointype, EndType endtype);

Adds a **Path** to a ClipperOffset object in preparation for offsetting.

Any number of paths can be added, and each has its own JoinType and EndType. All 'outer' Paths must have the same orientation, and any 'hole' paths must have reverse orientation. *Closed* paths must have at least 3 vertices. *Open* paths may have as few as one vertex. Open paths can only be offset with positive deltas.

EndType, JoinType, Path

∆ Home	ClipperLib	ClipperOffset	Clipper 🏡
---------------	------------	---------------	-----------

ClipperOffset.AddPaths

Del.» procedure AddPaths(const Paths: TPaths; JoinType: TJoinType; EndType: TEndType); C++ » void AddPaths(const Paths& paths, JoinType jointype, EndType endtype); C# » public void AddPaths(Paths paths, JoinType jointype, EndType endtype);

Adds **Paths** to a ClipperOffset object in preparation for offsetting.

Any number of paths can be added, and each path has its own JoinType and EndType. All 'outer' Paths must have the same orientation, and any 'hole' paths must have reverse orientation. *Closed* paths must have at least 3 vertices. *Open* paths may have as few as one vertex. Open paths can only be offset with positive deltas.

EndType, JoinType, Paths

∆ Home	ClipperLib	ClipperOffset	Clipper 🏡
---------------	------------	---------------	-----------

ClipperOffset.Clear

Del.» procedure Clear; C++ » void Clear(); C# » public void Clear();

This method clears all paths from the ClipperOffset object, allowing new paths to be assigned.

∆ Home	ClipperLib	ClipperOffset	Clipper 🏡
---------------	------------	---------------	-----------

ClipperOffset.Constructor

```
Del.» constructor Create(MiterLimit: Double = 2;
RoundPrecision: Double = 0.25);
C++ » ClipperOffset( double miterLimit = 2.0,
double roundPrecision = 0.25);
```

```
C# >> public ClipperOffset( double miterLimit =
2.0, double roundPrecision = 0.25);
```

The ClipperOffset constructor takes 2 optional parameters: MiterLimit and ArcTolerance. Thes two parameters corresponds to properties of the same name. MiterLimit is only relevant when JoinType is jtMiter, and ArcTolerance is only relevant when JoinType is jtRound or when EndType is etOpenRound.

ArcTolerance, MiterLimit

∆ Home	ClipperLib	ClipperOffset	Clipper 🏡
---------------	------------	---------------	-----------

ClipperOffset.Execute

```
Del.» procedure Execute(out solution: TPaths;
Delta: Double); overload;
C++ » void Execute(Paths& solution, double
delta);
C# » public void Execute(ref Paths solution,
double delta);
Del.» procedure Execute(out PolyTree: TPolyTree;
Delta: Double); overload;
C++ » void Execute(PolyTree& polytree, double
delta);
C# » public void Execute(ref PolyTree polytree,
double delta);
```

This method takes two parameters. The first is the structure that receives the result of the offset operation (either a PolyTree or a Paths structure). The second parameter is the amount to which the supplied paths will be offset. Negative delta values shrink polygons and positive delta expand them.

This method can be called multiple times, offsetting the same paths by different amounts (ie using different deltas).



```
#include "clipper.hpp"
using namespace ClipperLib;
int main()
{
 Path subj;
 Paths solution;
 subj <<
    IntPoint(348,257) << IntPoint(364,148) << IntPoint(362,148) <<</pre>
    IntPoint(326,241) << IntPoint(295,219) << IntPoint(258,88) <<
    IntPoint(440,129) << IntPoint(370,196) << IntPoint(372,275);</pre>
 ClipperOffset co;
 co.AddPath(subj, jtRound, etClosedPolygon);
 co.Execute(solution, -7.0);
 //draw solution ...
 DrawPolygons(solution, 0x4000FF00, 0xFF009900);
}
```

∆ Home	ClipperLib	ClipperOffset	Clipper 🏡
---------------	------------	---------------	-----------

ClipperOffset.ArcTolerance

```
Del.» property ArcTolerance: double; //read and
write
C++ » double ArcTolerance;
C# » public double ArcTolerance {get; set;}
```

Firstly, this field/property is only relevant when JoinType = jtRound and/or EndType = etRound.

Since flattened paths can never perfectly represent arcs, this field/property specifies a maximum acceptable imprecision ('tolerance') when arcs are approximated in an offsetting operation. Smaller values will increase 'smoothness' up to a point though at a cost of performance and in creating more vertices to construct the arc.

The default ArcTolerance is 0.25 units. This means that the maximum distance the flattened path will deviate from the 'true' arc will be no more than 0.25 units (before rounding).

Reducing tolerances below 0.25 will **not** improve smoothness since vertex coordinates will still be rounded to integer values. The only way to achieve sub-integer precision is through coordinate scaling before and after offsetting (see example below).

It's important to make ArcTolerance a sensible fraction of the offset delta (arc radius). Large tolerances relative to the offset delta will produce poor arc approximations but, just as importantly, very small tolerances will substantially slow offsetting performance while providing unnecessary degrees of precision. This is most likely to be an issue when offsetting polygons whose coordinates have been scaled to preserve floating point precision.

Example: Imagine a set of polygons (defined in floating point coordinates) that is to be offset by 10 units using *round* joins, and the solution is to retain floating point precision up to at least 6 decimal places.

To preserve this degree of floating point precision, and given that Clipper

and ClipperOffset both operate on integer coordinates, the polygon coordinates will be scaled up by 10^8 (and rounded to integers) prior to offsetting. Both offset delta and ArcTolerance will also need to be scaled by this same factor. *If ArcTolerance was left unscaled at the default 0.25 units, every arc in the solution would contain a fraction of 44 THOUSAND vertices* while the final arc imprecision would be 0.25×10^{-8} units (ie once scaling was reversed). However, if 0.1 units was an acceptable imprecision in the final *unscaled* solution, then ArcTolerance should be set to $0.1 \times \text{scaling}_{\text{factor}} (0.1 \times 10^8)$. Now if scaling is applied equally to both ArcTolerance and to Delta Offset, then in this example the number of vertices (steps) defining each arc would be a fraction of 23.

The formula for the number of steps in a full circular arc is ... Pi / acos(1 - arc_tolerance / abs(delta))

offset_triginometry2

∆ Home	ClipperLib	ClipperOffset	Clipper 🏡
---------------	------------	---------------	-----------

ClipperOffset.MiterLimit

```
Del.» property MiterLimit: double; //read and
write
C++ » double MiterLimit;
C# » public double MiterLimit {get; set;}
```

This property sets the maximum distance *in multiples of delta* that vertices can be offset from their original positions before squaring is applied. (Squaring truncates a miter by 'cutting it off' at 1 × delta distance from the original vertex.)

The default value for MiterLimit is 2 (ie twice delta). This is also the smallest MiterLimit that's allowed. If mitering was unrestricted (ie without any squaring), then offsets at very acute angles would generate unacceptably long 'spikes'.

An example of an offsetting 'spike' at a narrow angle that's a consequence of using a large MiterLimit (25) ...

JoinType

[™]Home	& ClipperLib	[⊷] PolyNode	Clipper 🏠
-------------------------	--------------	-----------------------	-----------

PolyNode.GetNext

```
Del.» function GetNext: TPolyNode;
C++ » PolyNode* GetNext();
C# » public PolyNode GetNext();
```

The returned Polynode will be the first child if any, otherwise the next sibling, otherwise the next sibling of the Parent etc.

A PolyTree can be traversed very easily by calling GetFirst() followed by GetNext() in a loop until the returned object is a null pointer ...

```
PolyTree polytree;
//call to Clipper.Execute method here which fills 'polytree'
PolyNode* polynode = polytree.GetFirst();
while (polynode)
{
   //do stuff with polynode here
   polynode = polynode->GetNext();
}
```

PolyTree.GetFirst

[™]Home	& ClipperLib	[⊷] PolyNode	Clipper 🏠
-------------------------	--------------	-----------------------	-----------

PolyNode.ChildCount

```
Del.» property ChildCount: Integer; //read only
```

```
C++ » ChildCount(); //read only
```

```
C# >> public int ChildCount; //read only
```

Returns the number of PolyNode Childs directly owned by the PolyNode object.
Childs

[™]Home	& ClipperLib	[⊷] PolyNode	Clipper 🏠
-------------------------	--------------	-----------------------	-----------

PolyNode.Childs

Del.» property Childs[index: Integer]: TPolyNode; //read only
C++ » std::vector < PolyNode* > Childs;//public
field
C# » public List < PolyNode > Childs; //read
only property

A read-only list of PolyNode.

Outer PolyNode childs contain hole PolyNodes, and hole PolyNode childs contain nested outer PolyNodes.

ChildCount

[™]Home	& ClipperLib	[⊷] PolyNode	Clipper 🏠
-------------------------	--------------	-----------------------	-----------

PolyNode.Contour

Del.» property Contour: TPath; //read only
C++ » Path Contour; //public field
C# » public Path Contour; //read only property

Returns a path list which contains any number of vertices.

[™]Home	& ClipperLib	[⊷] PolyNode	Clipper 🏠
-------------------------	--------------	-----------------------	-----------

PolyNode.IsHole

```
Del.» IsHole: Boolean; //read only
C++ » bool IsHole; //field
C# » public bool IsHole; //read only property
```

Returns true when the PolyNode's polygon (Contour) is a hole.

Children of *outer* polygons are always *holes*, and children of *holes* are always (nested) *outer* polygons.

The IsHole property of a **PolyTree** object is undefined but its children are always top-level *outer* polygons.

Overview, Contour, PolyTree

[™]Home	& ClipperLib	[⊷] PolyNode	Clipper 🏠
-------------------------	--------------	-----------------------	-----------

PolyNode.IsOpen

```
Del.» IsOpen: Boolean; //read only
C++ » bool IsOpen; //field
C# » public bool IsOpen; //read only property
```

Returns true when the PolyNode's Contour results from a clipping operation on an open contour (path). Only top-level PolyNodes can contain open contours.

Contour

[™]Home	& ClipperLib	[⊷] PolyNode	Clipper 🏠
-------------------------	--------------	-----------------------	-----------

PolyNode.Parent

```
Del.» Parent: TPolyNode; //read only
C++ » PolyNode* Parent; //field
C# » public PolyNode Parent; //read only
property
```

Returns the parent PolyNode.

The PolyTree object (which is also a PolyNode) does not have a parent and will return a null pointer.

AdHome ♦ClipperLib PolyTree Clipper

PolyTree.Clear

```
Del.» procedure Clear;
C++ » void Clear();
C# » public void Clear();
```

This method clears any PolyNode children contained by PolyTree the object.

Clear does *not* need to be called explicitly. The Clipper.Execute method that accepts a PolyTree parameter will *automatically* clear the PolyTree object before propagating it with new PolyNodes. Likewise, PolyTree's destructor will also automatically clear any contained PolyNodes.

Clipper.Execute

AdHome ♦ClipperLib PolyTree Clipper

PolyTree.GetFirst

```
Del.» function GetFirst: TPolyNode;
C++ » PolyNode* GetFirst();
C# » public PolyNode GetFirst();
```

This method returns the first outer polygon contour if any, otherwise a null pointer.

This function is almost equivalent to calling Childs[0] except that when a PolyTree object is empty (has no children), calling Childs[0] would raise an out of range exception.

PolyNode.GetNext, PolyNode.ChildCount, PolyNode.Childs

AdHome ♦ClipperLib PolyTree Clipper

PolyTree.Total

```
Del.» property Total: Integer; //read only
C++ » Total(); //read only
C# » public int Total; //read only
```

Returns the total number of PolyNodes (polygons) contained within the PolyTree. This value is not to be confused with ChildCount which returns the number of immediate children only (Childs) contained by PolyTree.

PolyNode.ChildCount, PolyNode.Childs

Overview

The **Clipper Library** performs **clipping**, and **offsetting** of both lines and polygons.

A number of features set Clipper apart from other clipping libraries:

- it accepts all types of polygons including self-intersecting ones
- it supports multiple polygon filling rules (EvenOdd, NonZero, Positive, Negative)
- it's very fast relative to other libraries
- it's numerically robust
- it also performs line and polygon offsetting
- it's free to use in both freeware and commercial applications

Current Version: 6.2.0

Author & copyright:

Angus Johnson. Copyright © 2010-2014 License, terms and conditions: Boost Software License

Terminology:

• Clipping:

commonly refers to the process of cutting away from a set of 2dimensional geometric shapes those parts that are outside a rectangular '*clipping*' window. This can be achieved by *intersecting*



subject paths (lines and polygons) with a clipping rectangle. In a more general sense, the clipping window need not be rectangular but can be any type of polygon, even multiple polygons. Also, while clipping typically refers to an *intersection* operation, in this documentation it will refer to any one of the four boolean operations (intersection, union, difference and exclusive-or).

- **Path:** is an ordered sequence of vertices defining a single geometric contour that's either a line (an open path) or a polygon (a closed path).
- **Line:** or polyline is an *open* path containing 2 or more vertices.

• **Polygon:** commonly refers to a two-dimensional region bounded by an outer non-intersecting closed contour. That region may also contain a number of 'holes'. In this documentation however, polygon will simply refer to a *closed* path.

• **Contour:** synonymous with path.

• **Hole:** is a closed region within a polygon that's not part of the polygon. A 'hole polygon' is a closed path that forms the outer boundaries of a hole.

• **Polygon Filling Rule:** the **filling rule**, together with a list of closed paths, defines those regions (bounded by paths) that are *inside* (ie regions 'brush filled' in a graphical display) and those which are *outside* (ie 'holes').

Distribution package contents:

The ZIP package contains the Clipper library's source code, a Windows CHM help file, HTML help, and a number of compiled demo applications (with full source code). The library was initially written in **Delphi Pascal** (and compiles with Delphi version 7 or above) but now contains **C++**, **C#** and **Python** translations too. The library's source code in each language is about 5000 lines. The Delphi code contains reasonably extensive comments, but comments are fewer in the C++ and C# code. The included sample applications show how Clipper can be used with the different languages using a number of graphics display libraries including **- AGG, Cairo, OpenGL, Graphics32, GDI+** and **SVG**.

Download Link:

SourceForge

References:

The Library is based on but significantly extends Bala Vatti's polygon clipping algorithm as described in "A generic solution to polygon clipping", Communications of the ACM, Vol 35, Issue 7 (July 1992) pp 56-63.

A section in "Computer graphics and geometric modeling: implementation and algorithms" by By Max K. Agoston (Springer, 2005) discussing *Vatti Polygon Clipping* was also helpful in creating the initial Clipper implementation.

The paper titled "Polygon Offsetting by Computing Winding Numbers" by Chen & McMains (Paper no. DETC2005-85513, ASME 2005. Pages 565-575) contains helpful discussion on the complexities of polygon offsetting together with some solutions.

Source, License, Clipper, ClipperOffset, ClipType, PolyFillType

🖆 Home	ClipperLib
--------	------------

4

ClipperBase

ClipperBase is an abstract base class for **Clipper**. A ClipperBase object should not be instantiated directly.

Reference

Methods In ClipperBase:

AddPath AddPaths Clear GetBounds

Clipper

🖆 Home	ClipperLib
--------	------------

4

Clipper

Hierarchy

| ClipperBase

The **Clipper** class encapsulates boolean operations on polygons (intersection, union, difference and XOR), which is also called polygon clipping.

Input polygons, both *subject* and *clip* sets, are passed to a Clipper object by its AddPath and AddPaths methods, and the clipping operation is performed by calling its Execute method. Multiple boolean operations can be performed on the same input polygon sets by repeat calls to Execute.

Reference

Methods	Properties
In Clipper:	
Constructor	PreserveCollinear
Execute	ReverseSolution
	StrictlySimple
	ZFillFunction
In ClipperBa	se:
AddPath	
AddPaths	
Clear	
GetBounds	

Overview, ClipType
🖆 Home	ClipperLib
--------	------------

4

ClipperOffset

The ClipperOffset class encapsulates the process of offsetting (inflating/deflating) both open and closed paths using a number of different join types and end types.

(This class replaces the now deprecated OffsetPaths function which was less flexible.)

Preconditions for offsetting:

1. The orientations of closed paths must be consistent such that outer polygons share the same orientation, and any holes have the opposite orientation (ie non-zero filling). Open paths must be oriented with closed outer polygons.

2. Polygons must not self-intersect.

Limitations:

When offsetting, small artefacts may appear where polygons overlap. To avoid these artefacts, offset overlapping polygons separately.



#include "clipper.hpp"
...
using namespace ClipperLib;
int main()
{

```
Path subj;
Paths solution;
subj <<
    IntPoint(348,257) << IntPoint(364,148) << IntPoint(362,148) <<
    IntPoint(326,241) << IntPoint(295,219) << IntPoint(258,88) <<
    IntPoint(440,129) << IntPoint(370,196) << IntPoint(372,275);
    ClipperOffset co;
    co.AddPath(subj, jtRound, etClosedPolygon);
    co.Execute(solution, -7.0);
    //draw solution ...
    DrawPolygons(solution, 0x4000FF00, 0xFF009900);
}
```

Reference

Methods	Properties	
In ClipperOffset:		
AddPath	ArcTolerance	
AddPaths	MiterLimit	
Clear		
Constructor		
Execute		

ՃHome	ClipperLib
-------	------------

4

PolyNode

PolyNodes are encapsulated within a **PolyTree** container, and together provide a data structure representing the parent-child relationships of polygon contours returned by Clipper's **Execute** method.

A **PolyNode** object represents a single polygon. Its IsHole property indicates whether it's an outer or a hole. PolyNodes may own any number of PolyNode children (**Childs**), where children of outer polygons are holes, and children of holes are (nested) outer polygons.

Reference

Methods	Properties	
In PolyNode:		
GetNext	ChildCount	
	Childs	
	Contour	
	IsHole	
	IsOpen	
	Parent	

See Also

Overview, Clipper.Execute, PolyTree

ՃHome	ClipperLib
-------	------------

4

PolyTree

Hierarchy

| PolyNode

PolyTree is intended as a **read-only** data structure that should only be used to receive *solutions* from clipping and offsetting operations. It's an alternative to the Paths data structure which also receives these solutions. PolyTree's two major advantages over the *Paths* structure are: it properly represents the **parent-child relationships** of the returned polygons; it differentiates between **open** and **closed** paths. However, since PolyTree is a more complex structure than the *Paths* structure, and since it's more computationally expensive to process (the Execute method being roughly 5-10% slower), it should used only be when parent-child polygon relationships are needed, or when open paths are being 'clipped'.

An empty PolyTree object can be passed as the *solution* parameter in **Clipper.Execute** and in **ClipperOffset.Execute**. Once the clipping or offseting operation is completed, the method returns with the PolyTree structure filled with data representing the solution.

A **PolyTree** object is a container for any number of **PolyNode** children, with each contained PolyNode representing a single polygon contour (either an outer or hole polygon). PolyTree itself is a specialized PolyNode whose immediate children represent the top-level *outer* polygons of the solution. (Its own Contour property is always empty.) The contained top-level PolyNodes may contain their own PolyNode children representing hole polygons that may also contain children representing nested outer polygons etc. Children of *outers* will always be *holes*, and children of *holes* will always be *outers*.

PolyTrees can also contain **open** paths. Open paths will always be represented by *top level* PolyNodes. Two functions are provided to quickly separate out *open* and *closed* paths from a polytree - **OpenPathsFromPolyTree** and **ClosedPathsFromPolyTree**.



polytree:

```
Contour = ()
ChildCount = 1
Childs[0]:
  Contour = ((10,10),(100,10),(100,100),(
  IsHole = False
  ChildCount = 1
  Childs[0]:
    Contour = ((20,20),(20,90),(90,90),(9
    IsHole = True
    ChildCount = 2
    Childs[0]:
       Contour = ((30,30),(50,30),(50,50)
       IsHole = False
       ChildCount = 0
    Childs[1]:
       Contour = ((60,60),(80,60),(80,80)
       IsHole = False
       ChildCount = 0
```

Reference

Methods	Properties	
In PolyTree:		
Clear	Total	
GetFirst		
In PolyNode:		
GetNext	ChildCount	
	Childs	
	Contour	
	IsHole	
	IsOpen	
	Parent	

See Also

Overview, Clipper.Execute, ClipperOffset.Execute, PolyNode, ClosedPathsFromPolyTree, OpenPathsFromPolyTree, Paths

ClipperLib

Filenames: clipper.pas; clipper.hpp and clipper.cpp; clipper.cs

Namespace: ClipperLib

Contents

Types	Classes	Functions
CInt	Clipper	Area
ClipType	ClipperBase	CleanPolygon
EndType	ClipperOffset	CleanPolygons
InitOptions	PolyNode	ClosedPathsFromPolyTree
IntPoint	PolyTree	MinkowskiDiff
IntRect		MinkowskiSum
JoinType		OffsetPaths
Path		OpenPathsFromPolyTree
Paths		Orientation
PolyFillType		PointInPolygon
PolyType		PolyTreeToPaths
ZFillCallback		ReversePath
		ReversePaths
		SimplifyPolygon
		SimplifyPolygons