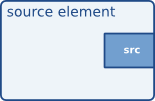
**What are elements?**

For the application programmer, elements are best visualized as black boxes. On the one end, you might put something in, the element does something with it and something else comes out at the other side. For a decoder element, for example, you'd put in encoded data, and the element would output decoded data.

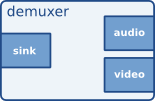
**Source elements:**Source elements generate data for use by a pipeline



Source elements do not accept data, they only generate data.

**Sink elements:**Sink elements are end points in a media pipeline. They accept data but do not produce anything. Disk writing, soundcard playback, and video output would all be implemented by sink elements





## Creating a GstElement:

The simplest way to create an element is to use [gst\_element\_factory\_make ()](http://gstreamer.freedesktop.org/data/doc/gstreamer/stable/gstreamer/html/GstElementFactory.html" \l "gst-element-factory-make)

[GstElement](https://gstreamer.freedesktop.org/data/doc/gstreamer/stable/gstreamer/html/GstElement.html) \*

gst\_element\_factory\_make (const [gchar](https://gstreamer.freedesktop.org/usr/share/gtk-doc/html/glibglib-Basic-Types.html" \l "gchar) \*factoryname,

const [gchar](https://gstreamer.freedesktop.org/usr/share/gtk-doc/html/glibglib-Basic-Types.html" \l "gchar) \*name);

Create a new element of the type defined by the given element factory. If name is [NULL](https://gstreamer.freedesktop.org/usr/share/gtk-doc/html/glibglib-Standard-Macros.html" \l "NULL:CAPS), then the element will receive a guaranteed unique name, consisting of the element factory name and a number. If name is given, it will be given the name supplied.

new [GstElement](https://gstreamer.freedesktop.org/data/doc/gstreamer/stable/gstreamer/html/GstElement.html) or [NULL](https://gstreamer.freedesktop.org/usr/share/gtk-doc/html/glibglib-Standard-Macros.html" \l "NULL:CAPS) if unable to create element.

gst\_element\_factory\_make is actually a shorthand for a combination of two functions. A [GstElement](http://gstreamer.freedesktop.org/data/doc/gstreamer/stable/gstreamer/html/GstElement.html) object is created from a factory. To create the element, you have to get access to a [GstElementFactory](http://gstreamer.freedesktop.org/data/doc/gstreamer/stable/gstreamer/html/GstElementFactory.html) object using a unique factory name. This is done with [gst\_element\_factory\_find ()](http://gstreamer.freedesktop.org/data/doc/gstreamer/stable/gstreamer/html/GstElementFactory.html" \l "gst-element-factory-find).

g\_object\_get(G\_OBJECT (element), "name", &name, NULL);

is used to get the name of the element

You can use gst\_registry\_pool\_feature\_list (GST\_TYPE\_ELEMENT\_FACTORY) to get a list of all the element factories that GStreamer knows about.

## Linking elements:

By linking these three elements, we have created a very simple chain of elements. The effect of this will be that the output of the source element (“element1”) will be used as input for the filter-like element (“element2”). The filter-like element will do something with the data and send the result to the final sink element (“element3”).

int

main (int argc,

char \*argv[])

{

GstElement \*pipeline;

GstElement \*source, \*filter, \*sink;

/\* init \*/

gst\_init (&argc, &argv);

/\* create pipeline \*/

pipeline = gst\_pipeline\_new ("my-pipeline");

/\* create elements \*/

source = gst\_element\_factory\_make ("fakesrc", "source");

filter = gst\_element\_factory\_make ("identity", "filter");

sink = gst\_element\_factory\_make ("fakesink", "sink");

/\* must add elements to pipeline before linking them \*/

gst\_bin\_add\_many (GST\_BIN (pipeline), source, filter, sink, NULL);

/\* link \*/

if (!gst\_element\_link\_many (source, filter, sink, NULL)) {

g\_warning ("Failed to link elements!");

}

}

**you must add elements to a bin or pipeline before linking them, since adding an element to a bin will disconnect any already existing links. Also, you cannot directly link elements that are not in the same bin or pipeline; if you want to link elements or pads at different hierarchy levels, you will need to use ghost pads**

GST\_STATE\_NULL:this is the default state. No resources are allocated in this state, so, transitioning to it will free all resources. The element must be in this state when its refcount reaches 0 and it is freed.

GST\_STATE\_READY: in the ready state, an element has allocated all of its global resources, that is, resources that can be kept within streams. You can think about opening devices, allocating buffers and so on. However, the stream is not opened in this state, so the stream positions is automatically zero. If a stream was previously opened, it should be closed in this state, and position, properties and such should be reset.

GST\_STATE\_PAUSED: in this state, an element has opened the stream, but is not actively processing it. An element is allowed to modify a stream's position, read and process data and such to prepare for playback as soon as state is changed to PLAYING, but it is not allowed to play the data which would make the clock run. In summary, PAUSED is the same as PLAYING but without a running clock.

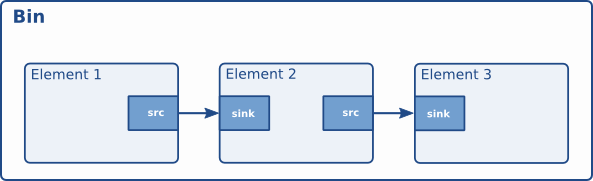
GST\_STATE\_PLAYING: in the PLAYING state, an element does exactly the same as in the PAUSED state, except that the clock now runs.

**Bins:**

A bin is a container element. You can add elements to a bin. Since a bin is an element itself, a bin can be handled in the same way as any other element

Bins allow you to combine a group of linked elements into one logical element. You do not deal with the individual elements anymore but with just one element, the bin. We will see that this is extremely powerful when you are going to construct complex pipelines since it allows you to break up the pipeline in smaller chunks.

The bin will also manage the elements contained in it. It will perform state changes on the elements as well as collect and forward bus messages.



**Creating a bin:**

Bins are created in the same way that other elements are created, i.e. using an element factory. There are also convenience functions available (gst\_bin\_new () and gst\_pipeline\_new ()). To add elements to a bin or remove elements from a bin, you can use gst\_bin\_add () and gst\_bin\_remove (). Note that the bin that you add an element to will take ownership of that element. If you destroy the bin, the element will be dereferenced with it. If you remove an element from a bin, it will be dereferenced automatically.

int

main (int argc,

char \*argv[])

{

GstElement \*bin, \*pipeline, \*source, \*sink;

/\* init \*/

gst\_init (&argc, &argv);

/\* create \*/

pipeline = gst\_pipeline\_new ("my\_pipeline");

bin = gst\_bin\_new ("my\_bin");

source = gst\_element\_factory\_make ("fakesrc", "source");

sink = gst\_element\_factory\_make ("fakesink", "sink");

/\* First add the elements to the bin \*/

gst\_bin\_add\_many (GST\_BIN (bin), source, sink, NULL);

/\* add the bin to the pipeline \*/

gst\_bin\_add (GST\_BIN (pipeline), bin);

/\* link the elements \*/

gst\_element\_link (source, sink);

[..]

}

There are various functions to lookup elements in a bin. The most commonly used are gst\_bin\_get\_by\_name () and gst\_bin\_get\_by\_interface (). You can also iterate over all elements that a bin contains using the function gst\_bin\_iterate\_elements ()

# **Bus:**

A bus is a simple system that takes care of forwarding messages from the streaming threads to an application in its own thread context. The advantage of a bus is that an application does not need to be thread-aware in order to use GStreamer, even though GStreamer itself is heavily threaded.

Every pipeline contains a bus by default, so applications do not need to create a bus or anything. The only thing applications should do is set a message handler on a bus, which is similar to a signal handler to an object. When the mainloop is running, the bus will periodically be checked for new messages, and the callback will be called when any message is available.

## **Pads:**

A pad type is defined by two properties: its direction and its availability. As we've mentioned before, GStreamer defines two pad directions: source pads and sink pads. This terminology is defined from the view of within the element: elements receive data on their sink pads and generate data on their source pads. Schematically, sink pads are drawn on the left side of an element, whereas source pads are drawn on the right side of an element.

# Buffers and Events:

# The data flowing through a pipeline consists of a combination of buffers and events. Buffers contain the actual media data. Events contain control information, such as seeking information and end-of-stream notifiers.

# Buffers contain the data that will flow through the pipeline you have created. A source element will typically create a new buffer and pass it through a pad to the next element in the chain. When using the GStreamer infrastructure to create a media pipeline you will not have to deal with buffers yourself; the elements will do that for you.

**gst\_init (&argc, &argv);**

his must always be your first GStreamer command. Among other things, gst\_init():

* Initializes all internal structures
* Checks what plug-ins are available
* Executes any command-line option intended for GStream

/\* Build the pipeline \*/

pipeline = gst\_parse\_launch ("playbin uri=https://www.freedesktop.org/software/gstreamer-sdk/data/media/sintel\_trailer-480p.webm", NULL);

### **gst\_parse\_launch:**

### GStreamer is a framework designed to handle multimedia flows. Media travels from the “source” elements (the producers), down to the “sink” elements (the consumers), passing through a series of intermediate elements performing all kinds of tasks. The set of all the interconnected elements is called a “pipeline”.

### In GStreamer you usually build the pipeline by manually assembling the individual elements, but, when the pipeline is easy enough, and you do not need any advanced features, you can take the shortcut: gst\_parse\_launch().

### This function takes a textual representation of a pipeline and turns it into an actual pipeline, which is very handy.

### **Playbin:**

### So, what kind of pipeline are we asking gst\_parse\_launch()to build for us? Here enters the second key point: We are building a pipeline composed of a single element called playbin.

### playbin is a special element which acts as a source and as a sink, and is a whole pipeline. Internally, it creates and connects all the necessary elements to play your media, so you do not have to worry about it.

**gst\_element\_set\_state (pipeline, GST\_STATE\_PLAYING);**