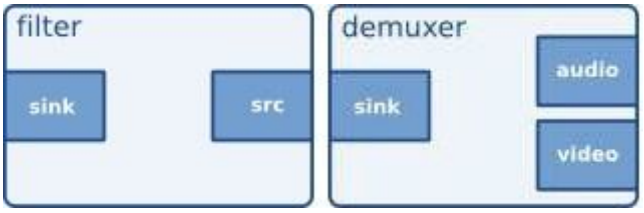
element

- GStreamer에 있어 가장 중요한 객체. 연결된 elements들의 체인을 만들 수도 있다. 하나의 element는 하나의 특정 기능을 가질 수 있음.
- Source element
  오직 파이프라인을 사용해서 데이터를 생성해 내는 역할. 디스크, 사운드 카드로 부터 읽는 것을 예로 들 수 있다. (오직 source pad만 존재)

- Filters, convertors, demuxers, muxers and codecs
  - input, output pad 모두 가지고 있다.
  - input pad를 통해 데이터를 전송 받고 output pad를 통해 데이터를 내보낸다.
  - 1개 이상의 source pad와 sink pad를 가질 수 있다.
  - video demuxer의 경우 1개의 sink pad와 여러개의 source 패드를 가지고 있는 반면에 decoder의 경우 1개의 sink pad와 source pad만 가진다.



- Sink element
  - Sink element는 pipeline의 end point를 지칭한다.(autovideosink)
  - 데이터는 받을 수 있지만 따로 생성하지는 못한다.
- Creating a GStElement
  - element를 생성하는 가장 쉬운 방법은 **gst_element_factory_make()** 함수를 이용하는 것이다.
  - **gst_element_factory_make()**는 factory name과 생성하고자 하는 element 이름을 매개변수로 받고 있다.
  - element 이름은 후에 debug 용도로 표시될 수도 있다. 이름을 NULL로 입력할 경우엔 default 이름이 붙는다.
  - **gst_object_unref()**를 통해 element의 삭제를 진행한다. 해당 함수는 Reference count (메모리를 제어하는 방법 중 하나로, garbage collection, 어떤 한 동적 단위가 참조값을 가지고 이 단위 객체가 참조되면 참조값을 늘리고 참조한 다음 더이상 사용하지 않게 되면 참조값을 줄이면 된다.) 를 줄이는 함수(아래 코드에서는 객체의 개수 1개를 줄이는 역할)
  - 아래 코드 fakesrc라는 이름을 가진 element factory로 부터 source라는 element를 생성하는 과정을 보여주고 있으며 종료 후에는 반드시 element unref를 해야한다.

```c
#include <gst/gst.h>

int main(int argc, char* argv[])
{
  GstElement *element;

  /* init GStreamer */
  gst_init(&argc, &argc);

  /* create element */
  element = gst_element_factory_make("fakesrc", "source");
  if(!element){
    g_print("Failed to create element of type 'fakesrc'\n");
    return -1;
  }

  gst_object_unref(GST_OBJECT(element));
  return 0;
}
```

- **gst_element_factory_make()**함수는 두개의 기능이 숨겨져 있다. GstElement는 factory에 의해 생성되고, element를 생성하기 위해서는 우선 해당 factory name을 사용하고 있는 GstElementFactory에 접근해서 가져와야 한다.
- factory를 찾을 수 있도록 제공하는 함수가 **gst_element_factory_find()** 함수이고 해당 factory로 부터 element를 가져오는 함수가 **gst_element_factory_create()** 함수 이다.-> 두가지 기능이 함께 들어있는 게 **gst_element_factory_create()**

- using an element as a GObject
  - GstElement는 표준 GObject 프로퍼티를 사용하여 구현된 다양한 프로퍼티들을 가질 수 있다.
  - 모든 GstElement들은 부모 객체로부터 적어도 하나의 프로퍼티를 가지는데, 바로 "name” 프로퍼티이다.(위의 함수 그리고 gst_object_set_name(), **gst_object_get_name()** 함수를 통해서도 name 프로퍼티를 수정할 수 있다.)

Element States
element가 생성되었다고 해서 곧바로 작업을 수행하는 것은 아니다. element의 상태를 변경해야 할 필요가 있다, element는 총 4가지 상태로 나눌 수 있으며, 다음과 같다.

- GST_STATE_NULL
  - default 상태, 이 상태에서는 어떤 resource도 할당 받지 못한 상태이다.

    - reference count가 0 이거나 free 상태일 때 element는 반드시 NULL 상태를 유지해야 한다.
- GST_STATE_READY
    - ready 상태, element가 resource를 막 할당받은 상태이다. 이 말인 즉, buffer등의 resource가 Stream 내에 보존되고 있다는 말이다.
    - 하지만 이 상태는 Stream이 열려인지 않기에 Stream위치는 자동으로 0이 된다.
    - 만약 이전 상태에서 Stream이 열려 있었다면, 이 상태에서는 Stream이 닫혀야 되고, Stream 위치나 프로퍼티도 재설정 되어야 한다.
- GST_STATE_PAUSED
    - 이 상태에서는 Stream이 열려 있기는 하지만, 동작하고 있지는 않다.
    - element가 스트림의 위치를 변경하는 등의 상태 변화는 허락하지만, 데이터를 재생시키는 것은 허락하지 않는다.
    - clock이 동작하지 않는 다는 것을 빼면 PLAYING 상태와 동일하다.
- GST_STATE_PLAYING
    - PLAYING 상태. clock이 현재 동작하고 있다는 것을 제외하고는 PAUSE 상태와 동일하다.

- gst_element_set_state() 함수를 통하여 element의 상태를 변경할 수 있다. 만약 사용자가 element의 상태를 NULL에서 PLAYING으로 변경한다면, GStreamer 내부에서는 READY와 PAUSED 상태를 모두 거쳐 PLAYING 상태로 변경할 것이다.

  "pad-added" or "new-decoded-pad"등의 signal callback을 통해 이미 동작중인 pipeline에 동적으로 element를 추가하려고 한다면, **gst_element_set_state()**나 **gst_element_sync_state_with_parent()** 함수를 사용하여 해당 element의 상태를 변경해야 한다.

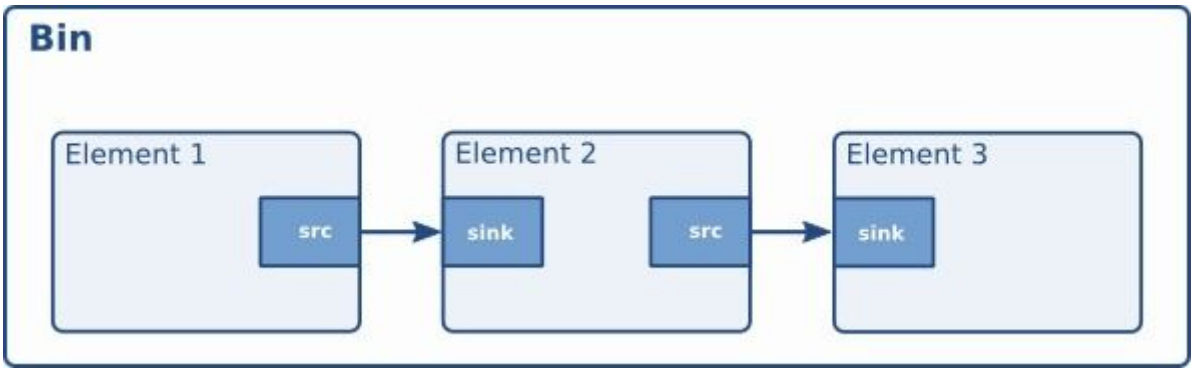
Linking elements

Data는 Source element에서 생성되어 0개 또는 다중의 filter element들을 거친 후 결국에는 sink element로 흐르게 된다.

```c
#include <gst/gst.h>

int main(int argc, char* argv[])
{
  GstElement* pipeline;
  GstElement* source, *filter, *sink;

  gst_init(&argc, &argv);

  /* createae pipeline */
  pipeline = gst_pipeline_new("my_pipeline");

  /* create element */
  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!");
  }

  .....
}
```

링크 시 중요한 사항은 bin 또는 pipeline에 element를 추가할 때, **이미 연결되어 있는 link에 대한 disconnect를 먼저 확인 해야한다. 다른 계층? 레벨에서 element 또는 pads를 연결하고자 할 때는 ghost pads를 사용해야 한다.**

Bin
Bin은 컨테이너 element라고 할 수 있음
새롭게 생성된 element를 Bin에 추가할 수 있으며, Bin 그 자체는 속해 있는 element들에 대한 처리가 가능하다. Bin에 대해 조금 더 자세히 설명하자면, 연결되어 있는 element들이 논리적으로 결합된 형태로 나타나는 것이 Bin이다. 각각의 element들을 처리할 필요 없이 하나의 element, 즉 Bin을 통해 처리가 가능하다.



- Bin 생성
  - Bin은 다른 element들을 생성할 때와 동일한 방식으로 생성된다.
  - element factory를 사용한다면, **gst_bin_new(), gst_pipeline_new()** 등과 같은 함수 사용이 더 편리할 것이다.
  - element를 bin에 추가하고 삭제하기 위해서는 **gst_bin_add(), gst_bin_remove()** 함수를 사용
  - Bin에 포함되어 있는 모든 element의 목록을 알고 싶을 때는 **gst_bin_get_list()**라는 함수를 이용한다.

```c
#include <gst/gst.h>

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);

  [...]
}
```

- Custom bins
  - 특정 업무를 수행하기 위한 용도로 Custom bin을 생성할 수도 있다.
  - 예를 들어 사용자가 Ogg나 vorbis decoder를 만들려고 한다면, 이는 상당히 어리석을 수 있다.(playbin2 element 사용하면 됨)
  - Custom bin은 플러그인 또는 XML description을 통해 생성될 수 있다.
  - https://gstreamer.freedesktop.org/documentation/plugin-development/?gi-language=c
- Bins manage states of their children
  - Bin은 자신에 포함되어 있는 모든 element들을 관리한다고 말했다.
  - 만약에 특정 상태를 만들기 위해 **gst_element_set_state()**라는 함수를 Bin 또는 pipeline에 사용한다면, 이는 Bin에 포함되어 있는 모든 element들에게 적용될 것이다. 이 말인 즉, 가장 상위의 pipeline의 상태만 start 하거나 stop 할 수 있다면 나머지 element들의 관리가 가능하다란 말이다.
  - 하지만 동작 중인 Bin 또는 pipeline에 추가가 되었을 때는 고려가 필요하다. 새롭게 추가된 element는 자동적으로 pipeline이나 Bin의 현재 상태로 전환되지 않기 때문이다.
  - 대신에 직접 gst_element_set_state() 혹은 **gst_element_sync_state_with_parent()** 함수를 사용하여 현재 동작 중인 pipeline에 추가로 element의 상태를 설정해주어야 한다.

Bus

Bus는 현재의 thread context 내에서 pipeline thread에서 application으로 메시지를 전송할 때 쓰이는 단순한 시스템이다. bus의 이점은 application이 GStreamer를 사용하기 위해 현재 동작하고 있는 Thread에 대해 전혀 알 필요가 없다는 것이다.

모든 pipeline은 기본적으로 bus를 포함하고 있으며, application은 bus를 따로 생성할 필요가 없다. application은 오직 message handler만 설정하면 된다.

mainloop가 돌 때는 bus는 주기적으로 새로운 메시지를 체크하고, 메시지가 이용가능하다고 판단되면 callback을 호출한다.

- bus를 가져오는 방법
  - GLib main loop를 실행시킨 후, main loop에서 bus에 새로운 메시지가 있는지 감시하는 방법 **gst_bus_add_watch() or gst_bus_add_signal_watch()**

    gst_bus_add_watch() : pipeline의 bus에 메시지 핸들러를 첨부시키는 함수이다. bus에 메시지가 들어올 때 만다 이 핸들러가 호출된다. 핸들러가 유지되기 위해서는 핸들러의 리턴은 TRUE가 되어야 한다. 리턴이 FALSE면 핸들러가 제거된다.
  - **gst_bus_peak(), gst_bus_poll()** 함수를 사용한다. 자기 자신이 메시지를 체크하는 방법

```c
#include <gst/gst.h>

static GMainLoop *loop;

static gboolean my_bus_callback (GstBus *bus, GstMessage *message, gpointer data)
{
  g_print ("Got %s message\n", GST_MESSAGE_TYPE_NAME (message));
  switch (GST_MESSAGE_TYPE (message)) {
    case GST_MESSAGE_ERROR: {
      GError *err;
      gchar *debug;
      gst_message_parse_error (message, &err, &debug);
      g_print ("Error: %s\n", err->message);
      g_error_free (err);
      g_free (debug);
      g_main_loop_quit (loop);
      break;
    }
    case GST_MESSAGE_EOS:
      /* end-of-stream */
      g_main_loop_quit (loop);
      break;
    default:
      /* unhandled message */
      break;
  }
  /* we want to be notified again the next time there is a message
  * on the bus, so returning TRUE (FALSE means we want to stop watching
  * for messages on the bus and our callback should not be called again)
  */
  return TRUE;
}

```

```c
gint main (gint argc, gchar *argv[])
{
  GstElement *pipeline;
  GstBus *bus;

  /* init */
  gst_init (&argc, &argv);

  /* create pipeline, add handler */
  pipeline = gst_pipeline_new ("my_pipeline");

  /* adds a watch for new message on our pipeline's message bus to
  * the default GLib main context, which is the main context that our
  * GLib main loop is attached to below
  */
  bus = gst_pipeline_get_bus (GST_PIPELINE (pipeline));
  gst_bus_add_watch (bus, my_bus_callback, NULL);
  gst_object_unref (bus);
  [..]

  /* create a mainloop that runs/iterates the default GLib main context
  * (context NULL), in other words: makes the context check if anything
  * it watches for has happened. When a message has been posted on the
  * bus, the default main context will automatically call our
  * my_bus_callback() function to notify us of that message.
  * The main loop will be run until someone calls g_main_loop_quit()
  */
  loop = g_main_loop_new (NULL, FALSE);
  g_main_loop_run (loop);

  /* clean up */
  gst_element_set_state (pipeline, GST_STATE_NULL);
  gst_object_unref (pipeline);
  g_main_loop_unref (loop);
  return 0;
}
```

gst_bus_add_watch() 함수를 사용한 후 switch를 사용하는 것 외에 gst_bus_add_signal_watch()함수를 사용할 수도 있다.

gst_bus_add_signal_watch() 함수 사용 시에는 아래와 같이 message type과 특정 callback 함수를 지정해 주어야 한다.

```
GstBus* bus;

bus = gst_pipeline_get_buf(GST_PIPELINE(pipeline));

gst_bus_add_signal_watch(bus);

g_signal_connect(bus, "message::error", G_CALLBACK(cb_message_error), NULL);

g_signal_connect(bus, "message::eos", G_CALLBACK(cb_message_eos), NULL);
```

Message types
미리 정의된 Message type이 있다. 또 추가적인 message도 정의 가능하다.
모든 message들은 message source와 type 그리고 timestamp를 가지고 있다. message source는 어떤 element가 message를 전달했는지를 알기 위해서 사용한다.

- Error, warning and information notifications
  - pipeline의 상태에 대해 사용자에게 알리는 용도로 사용되는 message
  - Error message는 치명적인 결함을 알리며, data의 흐름을 종료한다.
  - Warning은 치명적이지는 않지만, 문제를 일으킬 소지가 있다.
  - Information message는 문제가 발생하여 메시지를 보내는 것은 아님.
  - message들은 main error type과 message, 그리고 debug string이 있는 GError를 포함하고 있다.
  - **gst_message_parse_error()**, **_parse_warning()** 그리고 **_parse_info()**를 사용하여 GError를 parsing할 수 있다.
- End - of -stream notification
  - Stream이 끝났을 때 전송
  - application은 재생 목록에 있는 다음 음악으로 넘기기 위해 메시지를 이용할 수 있다.
  - message에서 가지고 있는 특별한 arg 는 없다.
- Tags
  - Stream에서 Metadata가 발견되었을 때 전송된다.
  - pipeline이 동작하는 동안 여러 차례 전송될 수 있다.
  - **gst_message_parse_tag()**를 통해 tags message를 파싱하고, 더 이상 사용할 필요가 없을 땐 **gst_tag_list_free()**함수로 자원을 해제한다.
- State-changes
  - 성공적으로 상태가 변경된 후에 전송
  - **gst_message_parse-state_changed()** 함수는 이전 상태와 새로 변경된 상태를 parsing하기 위해 사용된다.
- Buffering
  - 네트워크 스트림을 저장하는 동안에 message가 전송된다.
  - **gst_message_get_structure()** 함수에 의해 리턴된 구조체로부터 "buffer-percent" 프로퍼티를 추출함으로써 buffering의 진행 상태를 확인할 수 있다.
- Element messages

    - 특정 element가 특정 기능이 동작할 때, element message가 전송된다.
    - 예를 들어 QuickTime demuxer element는 Stream이 redirect명령을 포함하고 있을 때, 특정 경우에 대해 'redirect' element message를 전송할 수 있다.
- Application-specific message

Pad
Pad타입은 'direction'과 'availability' 두가지 프로퍼티에 의해 정의된다.
GStreamer 에서는 'source pad'와 'sink pad'라는 두 개의 pad direction을 정의한다. 또한 pad는 3개의 availability를 가진다. always, sometimes, on request.

always pad는 항상 존재하는 pad
simtimes pad는 어떤 특정 경우에는 존재하는 pad
on request pad는 application에 의해 요청이 들어왔을 때만 존재하는 pad를 말한다.

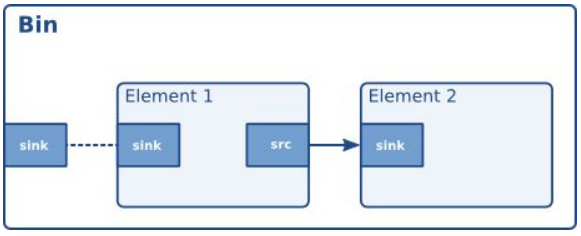
- Dynamic(sometime) pad
    - 모든 element들이 생성되었을 때부터 모든 pad를 지니고 있는 것은 아니다.
    - 예를 들어 Ogg demuxer element는 Stream이 검출 될 때만 dynamic pads를 생성하며, Stream이 종료될 때는 해당 pad를 제거하는 특징을 가진다.

```c
#include <gst/gst.h>
static void cb_new_pad (GstElement *element, GstPad *pad, gpointer data)
{
  gchar *name;
  name = gst_pad_get_name (pad);
  g_print ("A new pad %s was created\n", name);
  g_free (name);
  /* here, you would setup a new pad link for the newly created pad */
  [..]
}

int main (int argc, char *argv[])
{
  GstElement *pipeline, *source, *demux;
  GMainLoop *loop;

  /* init */
  gst_init (&argc, &argv);

  /* create elements */
  pipeline = gst_pipeline_new ("my_pipeline");
  source = gst_element_factory_make ("filesrc", "source");
  g_object_set (source, "location", argv[1], NULL);
  demux = gst_element_factory_make ("oggdemux", "demuxer");

  /* you would normally check that the elements were created properly */
  /* put together a pipeline */
  gst_bin_add_many (GST_BIN (pipeline), source, demux, NULL);
  gst_element_link_pads (source, "src", demux, "sink");

  /* listen for newly created pads */
  g_signal_connect (demux, "pad-added", G_CALLBACK (cb_new_pad), NULL);

  /* start the pipeline */
  gst_element_set_state (GST_ELEMENT (pipeline), GST_STATE_PLAYING);
  loop = g_main_loop_new (NULL, FALSE);
  g_main_loop_run (loop);
  [..]
}
```

        - **gst_element_set_state()** 또는 **gst_element_sync_state_with_parent()**함수 사용이 필요함.
- Request pads
    - element가 생성할 때 자동적으로 생성되는 것이 아니라, 요청이 있을 때만 pads가 생성된다.
    - input stream을 request로 생성된 여러 개의 output pads에게 복사해주는 element가 존재한다고 가정, application 입장에서는 stream의 복사가 필요할 때 마다 element로 부터 새로운 output pad만 요청하면 됨.
    - gst_element_get_request_pad()라는 함수는 element로부터 pad를 가져오는데 사용. 이렇게 request로 요청하여 pad를 가져오게 되면 다른 pad들과 호환성을 유지시킬 수 있다.
    - gst_element_get_compatible_pad()함수 역시 호환되는 pad를 요청할 때 사용되는 함수, gst_element_get_request_pad()와 gst_element_get_compatible_pad()

- Pad는 element의 input과 output을 나타냄. 이를 통해 element와 연결할 수가 있다. capability를 가지고 있음(특정 데이터의 흐름을 제한할 수 있다.)
- source pad - 데이터의 흐름을 밖으로 보낼 때
- sink pad - 데이터의 흐름을 element 내부로 가져올 때 사용된다.

Ghost pad



Bin 자체도 내부 element와 연결하거나 외부로 부터 데이터를 받기 위해서 존재하는 pad가 ghostpad이다. 위의 그림에서 element1의 sink pad는 bin의 sink pad의 역할도 함께 수행

```c
#inlcude <gst/gst.h>

int main(int argc, char* argv[])
{
  GstElement* bin, *sink;
  gstpad* pad;

  /* init */
  gst_init(&argc, &argv);

  /* create element, add to bin */
  sink = gst_element_factory_make("fakesink", "sink");
  bin = gst_bin_new("mybin");
  gst_bin_add(GST_BIN(bin), sink);

  /* add ghostpad */
  pad = gst_element_get_static_pad(sink, "sink");
  gst_element_add_pad(bin, gst_ghost_pad_new("sink", pad));
  gst_object_unref(GST_OBJECT(pad));

  [...]
}
```

Capability of pad
pad의 capability를 사용함으로 어떤 data type이 pad를 통해 흘러들어가고 나가는지를 확인할 수 있다.
pad의 capabilities는 GstCaps객체에 의해 기술된다. 내부적으로 GstCaps는 하나 이상의 GstStructure를 포함하고 있으며, 이 GstStructure는 하나의 미디어 타입에 대해 기술해 놓고 있다.
간단히 "vorbisdec" element를 예로, 해당 element에 대해 gst-inspect로 분석해 보면, source pad와 sink pad가 각각 한개씩 존재하는 것을 알 수 있다.

source pad는 raw audio mime 타입의 데이터("audio/x-raw-float")를 다음 element로 전송하고 있고 sink pad는 vorbis-encoded audio data 타입 ("audio/x-vorbis")을 수신하는 것을 알 수 있다. 또 source pad에서는 audio sample rate와 channel의 수, audio와 관련된 기타 property들을 가지고 있는 것을 알 수 있다.

- property
  해당 element의 특징, capability에 대한 추가 정보를 설명하기 위해 사용, rate, channel, endianeness 등등. property는 [key, value]형태로 되어 있는데 value의 type은 그 종류에 따라 다양하다.
- Basic type
  - G_type_INT : 정수값
  - G_TYPE_BOOLEAN : TRUE/FALSE
  - G_TYPE_STRING : UTF-8 기반의 문자열
  - GST_TYPE_FRACTION : 분수 값
- Range type
  - GST_TYPE_INT_RANGE : 정수 가능 형태의 범위 값. "vorbisdec" element의 경우 rate property가 8000~50000으로 property값이 정해진다.
  - GST_TYPE_FLOAT_RANGE : 부동 소수점 가능 형태의 범위 값.

    - GST_TYPE_FRACTION_RANGE : 분수 가능 형태의 범위 값.
- List type
    - GST_TYPE_LIST : 리스트 형태의 값. 예를 들어 주파수 대역이 44100Hz를 지원하는 capabilities의 경우 해당 값을 리스트 형태로 사용하면, 해당 값이 44100Hz가 될 수도 48000Hz가 될 수도 있다.
- Array Type
    - GST_TYPE_ARRAY : 배열 형태의 값, 배열 안에 있는 모든 item의 값들은 동일한 데이터 타입을 유지해야 한다.
    - GST_TPYE_LSIT와는 다르게 배열 내의 값들은 전체로써 해석된다. 이 말인즉, 44100Hz 와 48000Hz 값을 리스트에 넣으면 둘 중 하나로 인식되지만 배열로 넣으면 둘 다 인식 한다는 뜻

Capability
cap이라고도 불리며 pad에서 지원하고 있는 데이터의 type에 대해 기술하고 있다.
매뉴얼에서는 크게 4가지 방법을 제시하고 있다.

- Autoplugging
    - element가 자동으로 cap이 지원하는 pad를 연결할 수 있도록 한다.
- Compatibility detection
    - 두 개의 pads가 서로 연결되어 있다고 하면, GStreamer는 해당 pads가 동일한 미디어 타입을 지원하는 지를 검증할 수 있다.
    - cap을 통해 해당 검증을 가능케 해준다.
- Metadata
    - pad로부터 cap 정보를 읽음으로써, application은 현재 pad에 흐르고 있는 미디어 타입에 대한 정보를 제공할 수 있다.
    - pad는 하나 이상의 cap을 가지고 있으며, cap 여시 하나 이상의 GstStructure 집합을 가지고 있다.
    - 각각의 GstStructure는 field이름과(e.g. “width”) type으로 (e.g. G_TYPE_INT or GST_TYPE_INT_RANGE) 구성된 배열 형태로 구성되어 있다.
    - application은 cap 집합에 대해 query를 보냄으로서 property 각각에 대한 값을 가져올 수 있다.
    - **gst_cap_get_structure()**를 사용하여 cap으로부터 구조체를 가져올 수 있으며, **gst_caps_get_size()**를 통해 구조체의 개수를 알 수 있다.
- Filtering
    - application 은 현재 pad에 동작하는 미디어에 대한 제한을 걸기 위한 용도로 cap을 사용할 수 있다.
    - application은 video size를 설정하기 위해 "filtered caps"을 사용할 수 있다.
    - pipeline에 capsfilter element를 삽입하고, “caps" property를 설정함으로써 caps를 필터링할 수 있다.
    - 물론 **capsfilter를 사용하기 위해서는 자신만의 GstCaps를 만들어야한다. gst_caps_new_simple()**을 사용하여 GstCaps를 생성한다.
    - Caps filter는 대부분 audioconvert, aucioresample 등과 같은 converter element 뒤에 위치하게 된다.

```
gboolean link_ok;

GstCaps* caps;

caps = gst_caps_new_full(gst_structure_new("video/x-raw-yuv",

                                           "width", G_TYPE_INT, 384,

                                           "height", G_TYPE_INT, 288,

                                           "framerate", GST_TYPE_FRACTION, 25, 1,

                                           NULL),

                         gst_structure_new("video/x-raw-rgb",

                                           "width", G_TYPE_INT, 384,

                                           "height", G_TYPE_INT, 288,

                                           "framerate", GST_TYPE_FRACTION, 25, 1,

                                           NULL),

                         NULL);

link_ok = gst_element_link_filtered(element1, element2, caps);
```

Events

- upstream 혹은 downstream으로 전송될 때 보내지는 제어 객체
- upstream : element 간 뿐만 아니라 application과 element 간에도 사용, seek와 같은 stream의 상태 변화를 요청하기 위해 사용된다. application 입장에서 upstream이벤트를 주로 다룸
- downstream 이벤트는 stream 상태를 알려주며 seeking, fluches, end-of-stream notification등이 있다.
- upstream 이벤트는 element 간 뿐 아니라 application과 element 간에도 사용되며, seek과 같은 stream의 상태 변화를 요청하기 위해 사용된다.
- application 입장에서는 upstream이벤트를 자주 다룬다.

Tutorial 1

초기화 gst_init

- <gst/gst.h>를 include 해줘야 한다. 그 후에는 초기화가 필요하다.
- gst_init이라는 함수는 GStreamer 기반의 명령 라인 옵션을 parsing할 뿐만 아니라 비초기화된 라이브러리들을 초기화 하는 역할을 담당
- GST_VERSION_MAJOR, GST_VERSION_MINOR, GST_VERSION_MICRO 매크로를 사용하여 GStreamer 버전을 알 수 있다.

gstreamer에서 영상을 재생시키기 위해서는 source element와 sink element가 존재해야 하고 source에서 sink로 데이터가 흘러야 한다. 이 일련의 과정을 pipeline이라고 부른다.

- gst_parse_launch()를 이용해 자동으로 pipeline을 생성할 수도 일일이 구축해 생성할 수 있다.
- 수동
  - 일일이 생성하는 경우는 요소에 어떤 이벤트를 받고자 할 경우
  - source element에서 sink element로 연결이 이루어졌을 때, 이벤트를 잡아 어떤 처리를 하고 싶다. 이런 경우에 개발자가 일일이 개별 요소를 만들어 주어야 한다.
- 자동
  - 단지 영상 재상만을 목적으로 하는 경우라면 gst_parse_launch를 통해 알아서 파이프라인을 구축하게 하는 것이 간편
  - GstElement * gst_parse_launch(const gchar *pipeline_description, GError **error);

    playbin이라는 element가 멀티미디어가 위치한 uri를 받아서 그에 맞는 적절한 element를 생성하게 된다. 이렇게 생성된 playbin element를 통해 gst_parse_launch에서 해당 playbin을 읽어들여 그에 맞는 pipeline을 새롭게 생성

- gst_element_set_state(pipeline, GST_STATE_PLAYING);
  - 파이프라인을 재생시키려고 함
- bus = gst_element_get_bus(pipeline);

  msg = gst_bus_timed_pop_filtered(bus, GST_CLOCK_TIME_NONE,
  (GstMessageType)(GST_MESSAGE_ERROR | GST_MESSAGE_EOS));

  - 파이프 라인으로 부터 bus를 가져오고 bus의 이벤트가 에러이거나(GST_MESSAGE_ERROR) 파일의 끝(GST_MESSAGE_EOS)일 때까지 block시킴을 의미한다. 실제 예제 코드로 컴파일 시에는 int를 GstMessage Type으로 바꿀 수 없다는 형변환 에러가 발생하기 때문에 (GstMessage Type)이라고 명시적으로 형변환을 시켜주었다.
- if(msg != NULL){

  gst_message_unref(msg);

  }

  gst_object_unref(bus);

  gst_element_set_state(pipeline, GST_STATE_NULL);

gst_object_unref(pipeline);

- Cleanup 코드. bus로부터 반환된 메시지를 해제하고, bus객체도 해제한다.
- pipeline의 상태를 null로 바꾸어 주고 객체를 해제하면 모든 자원 반환이 완료된다.

Tutorial2
수동으로 파이프라인 구축하기

```
#include <gst/gst.h>

int main(int argc, char *argv[]){
    GstElement *pipeline, *source, *sink;
    GstBus *bus;
    GstMessage *msg;
    GstStateChangeReturn ret;

    gst_init(&argc, &argv);

    source = gst_element_factory_make("videotestsrc", "source");
    sink = gst_element_factory_make("autovideosink", "sink");

    pipeline = gst_pipeline_new("test-pipeline");

    if(!pipeline || !source || !sink){
        g_printerr("Not all elements could be created.\n");
        return -1;
    }

    gst_bin_add_many(GST_BIN(pipeline), source, sink, NULL);
    if(gst_element_link(source,sink) != TRUE){
        g_printerr("Elements could not be linked.\n");
        gst_object_unref(pipeline);
        return -1;
    }

    g_object_set(source, "pattern", 0, NULL);

    ret = gst_element_set_state(pipeline, GST_STATE_PLAYING);
    if(ret == GST_STATE_CHANGE_FAILURE){
        g_printerr("Unable to set the pipeline to the playing state.\n");
        gst_object_unref(pipeline);
        return -1;
    }

    bus = gst_element_get_bus(pipeline);
    msg = gst_bus_timed_pop_filtered(bus, GST_CLOCK_TIME_NONE, (GstMessageType)(GST_MESSAGE_ERROR | GST_MESSAGE_EOS));

    if(msg != NULL){
        GError *err;
        gchar *debug_info;

        switch(GST_MESSAGE_TYPE(msg)){
            case GST_MESSAGE_ERROR:
                gst_message_parse_error(msg, &err, &debug_info);
                g_printerr("Error received from element %s: %s\n", GST_OBJECT_NAME(msg->src), err->message);
                g_printerr("Debugging information: %s\n", debug_info ? debug_info : "none");
                g_clear_error(&err);
                g_free(debug_info);
                break;
            case GST_MESSAGE_EOS:
                g_print("End-Of-Stream reached.\n");
                break;
            default:
                g_printerr("Unexpected message received.\n");
                break;
        }
        gst_message_unref(msg);
    }



    gst_object_unref(bus);
    gst_element_set_state(pipeline, GST_STATE_NULL);
    gst_object_unref(pipeline);
    return 0;
}
```

- gst_init(&argc, &argv)
  초기화
- source = gst_element_factory_make(“videotestsrc”,”source”)
  sink = gst_element_factoy_make(“autovideosink”,”sink”)

    - 수동으로 파이프라인을 구축하기 위해 필요한 함수
    - 첫 번째 매개변수 : element type, 두 번째 매개변수 : custom name

- pipeline = gst_pipeline_new(“test-pipeline”)
    - 파이프라인 생성
- gst_bin_add_many(GST_BIN(pipeline), source, sink, NULL);

  ```
  if(gst_element_link(source,sink) != TRUE){

     g_printerr("Elements could not be linked.\n");

     gst_object_unref(pipeline);

     return -1;

  }
  ```

    - gst_bin_add_many 함수를 통해 pipeline에 element들을 추가한다. 만약 하나의 element만을 추가하고 싶을 경우 gst_bin_add 함수를 이용한다.
    - pipeline에 담은 element들을 서로 연결 - gst_element_link() 그리고 파라미터로 연결하고 자 할 element를 넘겨준다. **element간의 연결은 같은 bin(pipeline)에 있는 element 간에만 된다.**

- g_object_set(source,”pattern”, 0, NULL)
    - 대부분의 element는 속성 값을 가지고 있다. 속성 값은 g_object_get 함수로 읽을 수가 있고 g_object_set 함수로 설정할 수가 있다.
    - 파라미터로는 우선 바꾸고자 하는 속성 값을 지닌 element, 그리고 속성 이름과 바꾸려고 하는 값 그리고 마지막 값은 NULL을 전달해야 한다.

```
msg = gst_bus_timed_pop_filtered(bus, GST_CLOCK_TIME_NONE, (GstMessageType)
(GST_MESSAGE_ERROR | GST_MESSAGE_EOS));
if(msg != NULL){
   GError *err;
   gchar *debug_info;
   switch(GST_MESSAGE_TYPE(msg)){
      case GST_MESSAGE_ERROR:
         gst_message_parse_error(msg, &err, &debug_info);
         g_printerr("Error received from element %s: %s\n", GST_OBJECT_NAME(msg->src), err-
>message);
         g_printerr("Debugging information: %s\n", debug_info ? debug_info : "none");
         g_clear_error(&err);
         g_free(debug_info);
      case GST_MESSAGE_EOS:
         g_print("End-Of-Stream reached.\n");
         break;
      default:
         g_printerr("Unexpected message received.\n");
         break;
   }
   gst_message_unref(msg);
}
```

- switch 문에서 해당 결과 메시지를 받았을 때 이 결과 메시지가 에러인디 EOS인지를 판별한 후 출력해 준다.

tutorial-3.c

```
#include <gst/gst.h>

typedef struct _CustomData{
    GstElement *pipeline;
    GstElement *source;
    GstElement *convert;
    GstElement *sink;
}CustomData;

static void pad_added_handler(GstElement *src, GstPad *pad, CustomData *data);

int main(int argc, char *argv[]){
    CustomData data;
    GstBus *bus;
    GstMessage *msg;
    GstStateChangeReturn ret;
    gboolean terminate = FALSE;

    gst_init(&argc, &argv);

    data.source = gst_element_factory_make("uridecodebin", "source");
    data.convert = gst_element_factory_make("audioconvert", "convert");
    data.sink = gst_element_factory_make("autoaudiosink", "sink");

    data.pipeline = gst_pipeline_new("test-pipeline");

    if(!data.pipeline || !data.source || !data.convert || !data.sink){
        g_printerr("Not all elements could be created.\n");
        return -1;
    }

    gst_bin_add_many(GST_BIN(data.pipeline), data.source, data.convert, data.sink, NULL);
    if(!gst_element_link(data.convert, data.sink)){
        g_printerr("Elements could not be linked.\n");
        gst_object_unref(data.pipeline);
        return -1;
    }

    g_object_set(data.source, "uri", "http://docs.gstreamer.com/media/sintel_trailer-480p.webm", NULL);

    g_signal_connect(data.source, "pad-added", G_CALLBACK(pad_added_handler), &data);

    ret = gst_element_set_state(data.pipeline, GST_STATE_PLAYING);
    if(ret == GST_STATE_CHANGE_FAILURE){
        g_printerr("Unable to set the pipeline to the playing state.\n");
        gst_object_unref(data.pipeline);
        return -1;
    }

    bus = gst_element_get_bus(data.pipeline);
    do{
        msg = gst_bus_timed_pop_filtered(bus, GST_CLOCK_TIME_NONE, (GstMessageType)(GST_MESSAGE_STATE_CHANGED |
GST_MESSAGE_ERROR | GST_MESSAGE_EOS));
```

```
        if(msg != NULL){
            GError *err;
            gchar *debug_info;

            switch(GST_MESSAGE_TYPE(msg)){
                case GST_MESSAGE_ERROR:
                    gst_message_parse_error(msg,&err, &debug_info);
                    g_printerr("Error received from element %s: %s\n", GST_OBJECT_NAME(msg->src), err->message);
                    g_printerr("Debugging information: %s\n", debug_info ? debug_info : "none");
                    g_clear_error(&err);
                    g_free(debug_info);
                    terminate = TRUE;
                    break;
                case GST_MESSAGE_EOS:
                    g_print("End-Of-Stream reached.\n");
                    terminate = TRUE;
                    break;
                case GST_MESSAGE_STATE_CHANGED:
                    if(GST_MESSAGE_SRC(msg) == GST_OBJECT(data.pipeline)){
                        GstState old_state, new_state, pending_state;
                        gst_message_parse_state_changed(msg, &old_state, &new_state, &pending_state);
                        g_print("Pipeline state changed from %s to %s:\n", gst_element_state_get_name(old_state),
gst_element_state_get_name(new_state));
                    }
                    break;
                default:
                    g_printerr("Unexpected message received.\n");
                    break;
            }
            gst_message_unref(msg);
        }
    }while(!terminate);

    gst_object_unref(bus);
    gst_element_set_state(data.pipeline, GST_STATE_NULL);
    gst_object_unref(data.pipeline);
    return 0;
 }

 static void pad_added_handler(GstElement *src, GstPad *new_pad, CustomData *data){
    GstPad *sink_pad = gst_element_get_static_pad(data->convert, "sink");
    GstPadLinkReturn ret;
    GstCaps *new_pad_caps = NULL;
    GstStructure *new_pad_struct = NULL;
    const gchar *new_pad_type = NULL;

    g_print("Received new pad '%s' from '%s':\n", GST_PAD_NAME(new_pad), GST_ELEMENT_NAME(src));

    if(gst_pad_is_linked(sink_pad)){
        g_print(" We are already linked. Ignoring.\n");
        goto exit;
    }

    new_pad_caps = gst_pad_get_current_caps(new_pad);
    new_pad_struct = gst_caps_get_structure(new_pad_caps, 0);
    new_pad_type = gst_structure_get_name(new_pad_struct);
    if(!g_str_has_prefix(new_pad_type,"audio/x-raw")){
        g_print(" It has type '%s' which is not raw audio. Ignoring.\n", new_pad_type);
        goto exit;
    }
```

```
    ret = gst_pad_link(new_pad, sink_pad);
    if(GST_PAD_LINK_FAILED(ret)){
        g_print(" Type is '%s' but link failed.\n", new_pad_type);
    }else{
        g_print(" Link succeeded (type '%s').\n", new_pad_type);
    }

exit:
    if(new_pad_caps != NULL){
        gst_caps_unref(new_pad_caps);
    }
    gst_object_unref(sink_pad);
}
```

```
typedef struct _CustomData{
    GstElement *pipeline;
    GstElement *source;
    GstElement *convert;
    GstElement *sink;
}CustomData;
```

데이터 전달을 용이하게 하기 위해 customdata라는 구조체를 선언

```
data.source = gst_element_factory_make("uridecodebin", "source");
data.convert = gst_element_factory_make("audioconvert", "convert");
data.sink = gst_element_factory_make("autoaudiosink", "sink");
data.pipeline =  gst_pipeline_new("test-pipeline");
```

uridecodebin, audioconvert, autoaudiosink 타입의 element들을 생성하고, 파이프라인을 생성하였다.

- uridecodebin은 해당 uri에 위치한 미디어를 raw media로 디코딩 해주는 역할을 한다.
- uridecodebin을 만든 후 재생을 하게 되면, uridecodebin에서는 주어진 uri를 처리할 수 있는 source element를 선택하고 해당 source element를 decodebin에 연결함으로써 역할을 수행하게 된다.
- audioconvert는 audio를 다양한 포맷으로 변환해 주는 역할을 하며, integer to float 변환, width/depth변환, 채널 변형등을 가능하게 해준다.
- autoaudiosink는 해당 미디어에 대한 적절한 audio sink를 찾아주는 역할을 한다.

```
gst_bin_add_many(GST_BIN(data.pipeline), data.source, data.convert, data.sink,
NULL);
if(!gst_element_link(data.convert, data.sink)){
    g_printerr("Elements could not be linked.\n");
    gst_object_unref(data.pipeline);
    return -1;
}
```

- gst_bin_add_many(GST_BIN(data.pipeline), data.source, data.convert, data.sink, NULL)을 이용해 모든 element를 하나의 bin으로 묶어 준다.
- 여기서 주목해야 할 부분은 gst_element_link(data.convert, data.sink)이다. 이 부분에서는 source를 제외한 나머지 두 개만 연결하는 것을 볼 수 있다, 그 이유는 video가 아닌 audio만 다룰 것이기 때문에 따로 나누는 작업이 필요하다.

- g_object_set(data.source, "uri", "http://docs.gstreamer.com/media/sintel_trailer-480p.webm", NULL);
    - sourcedml uri 속성의 값을 입력함으로써 source element가 해당 데이터를 찾을 수 있도록 해준다.
- g_signal_connect(data.source, “pad-added”, G_CALLBACK(pad-added_handler), &data);
    - source element에서 pad가 만들어 졌을 때 인터럽트 신호를 받아서 pad_added_handler callback함수로 호출하라는 이야기다.
    - 대부분의 eleement들마다 pad가 존재하는데 이 pad라는 것이 element간의 연결할 때 필요한 요소이다. 가령 source element와 sink element가 연결된다 함은 source element의 src pad와 sink element의 sink pad가 연결된다는 것이다.
    - source element는 1개 이상의 src pad를 가질 수 있는데, 예를 들어 지금 보는 uridecodebin source element 같은 경우에는 video와 연결하기 위한 pad가 한개, audio와 연결하기 위한 pad가 한개 이렇게 2개의 pad를 가지고 있다.

- 소스에서 어떻게 각각의 pad를 구분지은 후 연결하는 지 알아봄
    - GstPad *sink_pad = gst_element_get_static_pad(data->convert, “sink”)
      위의 함수를 이용해 convert element로 부터 sink pad를 가져온다. sink pad를
      가져오는 목적은 source element의 src pad와 연결을 시키기 위함이다.
      GstPad는 source를 가지고 있으며, convert가 오디오만 되어있기 때문에 오디오 정보만 가져 온다.
    - if(gst_pad_is_linked(sink_pad)){
      g_print(“We are already linked. Ignoring \n”)
      goto exit;
      }
      convert element로 부터 추출한 sink pad가 현재 연결되어 있는지 여부를 확인한다,

    - new_pad_caps = gst_pad_get_current_caps(new_pad);//핸들러 만들 때 지정해놈
      new_pad_struct = gst_cap_get_structure(new_pad_caps,0);
      new_pad_type = get_structure_get_name(new_pad_struct);
      if(!g_str_has_prefix(new_pad_type, “audio/x-raw”)){
      g_print(“it has type ‘%s’ which is not raw audio. Ignoring \n”, new_pad_type);
      goto exit;

      }

      콜백으로 넘어온 pad의 type을 알기위한 코드

한번에 알 수 없기에 여러 과정을 거쳐야 한다. new_pad_type이 audio가 아니면 callback을 종료시킨다.

- `GstPadLinkReturn` ret = gst_pad_link(new_pad, sink_pad);
  최종 소스를 파이프라인에 연결한다.

Stream Seek
스트림 위치값을 얻기 위한 방법
GstQuery가 피요하며 이는 element나 pad에게 정보를 요청하기 위해 사용

- msg = gst_bus_timed_pop_filtered(bus, 100 * GST_MSECOND, (GstMessageType)(GST_MESSAGE_STATE_CHANGED | GST_MESSAGE_ERROR | GST_MESSAGE_EOS | GST_MESSAGE_DURATION));

  - timeout이 새롭게 등장 -> 10초 동안 아무런 메시지가 없으면 CHANGED, ERROR, EOS, DURATION // 도착하지 않으면 NULL을 리턴한다. 그리고 NULL이면 스트림 위치 값을 계산하며 NULL이 아니면 handle_message를 실행

- GstFormat fmt = GST_FORMAT_TIME;

  gint64 current = -1;

  if(!gst_element_query_position(data.playbin, fmt, &current)){

  g_printerr("Colud not query current position.\n");

  }

  - gst_element_query_position을 이용해 스트림 위치를 가져온다. 값은 0부터 스트림 길이까지 nanosecondas형태로 (GST_FORMAT_TIME) 리턴한다.
- if(!GST_CLOCK_TIME_IS_VALID(data.duration)){

  if(!gst_element_query_duration(data.playbin, fmt, &data.duration)){

  g_printerr("Colud not query current duration.\n");

  }

  }

  - 전체 스트림 길이를 가져온다. 매번 가져올 필요 없이 GST_CLOCK_TIME_IS_VALID란 조건을 줘서 duration이 GST_CLOCK_TIME_NONE(시간이 정의되지 않을 경우)일 경우에만 전체 스트림의 길이를 가져온다.
  -

- g_print("Position %" GST_TIME_FORMAT "/ %" GST_TIME_FORMAT "\r", GST_TIME_ARGS(current), GST_TIME_ARGS(data.duration));
    - h:m:s형태로 스트림 시간 데이터를 출력한다.
    - #define GST_TIME_FORM "u:%02u:%02u.%09u"
    -
- if(data.seek_enabled && !data.seek_done && current > 10 * GST_SECOND){

  g_print("\nReached 10s, performing seek...\n");

  gst_element_seek_simple(data.playbin, GST_FORMAT_TIME,(GstSeekFlags)(GST_SEEK_FLAG_FLUSH | GST_SEEK_FLAG_KEY_UNIT), 30 * GST_SECOND);

  data.seek_done = TRUE;

  }

    - 스트림의 시작으로부터 상대 위치를 찾을 때 gst_element_seek_simple
    - GST_FORMAT_TIME은 명시된 위치를 나타내기 위한 시간 포맷이며, SeekFlag 옵션을 제공ㅎ나다. Flag옵션의 종류는 아래와 같다.
    - GST_SEEK_FLAG_FLUSH

      현재 파이프라인 안에 존재하는 모든 데이터를 버린다. 만약 이 flag가 없다면 파이프 라인 끝에서 새로운 위치가 나타나기 전까지 이전 데이터가 계속 나타날 것이다.

    - GST_SEEK_FLAG_KEY_UNY

      가장 가까운 keyframe을 찾는다. 이 옵션은 빠르게 위치를 찾지감 정확도가 떨어질 수 있다.

    - GST_SEEK_FLAG_ACCURATE

      정확한 위치가 필요할 때 사용. 몇몇 포맷에 있어서는 느리게 동작할 수 있음

    - 파라미터로 찾고자 하는 시작위치를 지정할 수도 있음. nano second이기 때문에 GST_SECOND을 곱해주면 초로 바뀜

- case GST_MESSAGE_DURATION:

  data->duration = GST_CLOCK_TIME_NONE;

  break;

    - 스트림 길이가 변경되었을 때 case문에 진입하며 길이가 변경되면 data의 duration을 GST_CLOCK_TIME_NONE으로 설정해서 다시 한번 스트림 길이를 측정할 수 있다.
-

```
if(data->playing){
  GstQuery *query;
  gint64 start, end;
  query = gst_query_new_seeking(GST_FORMAT_TIME);
  if(gst_element_query(data->playbin, query)){
    gst_query_parse_seeking(query, NULL, &data->seek_enabled, &start, &end);
    if(data->seek_enabled){
      g_print("Seeking is ENABLED from %" GST_TIME_FORMAT " to %" GST_TIME_FORMAT "\n",
GST_TIME_ARGS(start), GST_TIME_ARGS(end));
    }else{
      g_print("Seeking is DISABLED for this stream.\n");
    }
  }else{
    g_printerr("Seeking query failed.");
  }
  gst_query_unref(query);
}
```

- 파이프 상태가 GST_STATE_PLAYING일 때 조건문에 진입한다. gst_query_new_seeking함수는 스트림의 seeking속성을 물어보기 위해서 새로 query객체를 생성한다. 이렇게 생성된 query객체를 gst_element_query 함수에 따라 파라미터로 넣음으로 playbin에 대한 seeking 속성 정보를 요청한다.
- gst_query_parse_seeking 함수는 query결과에 대해 파싱을 진행한다. seeking 가능 여부, seeking 시작 위치, seeking 끝 위치를 결과로 알려준다. 마지막으로 query 객체를 해제 해야한다.

basic-tutorial -8.c

Push, Pull 방식

엘리먼트의 pad는 push, pull 방식의 스케쥴링 모드를 지원합나다.

nnstreamer_sink_example_play.c

bins

- element들을 모아 놓은 하나의 container/ 일일이 element를 제어할 필요 없이 한번에 제어가 가능함. 즉 상태를 변경하면 bin 내부의 모든 element의 상태가 바뀌게 된다.

pipeline

- 위의 소스에서 싱크로 데이터가 흘러야 한다. 흘러가는 일련의 과정을 다음과 같이 명한다.

COMMUNICATION

- buffer
  ⇒ pipeline을 통해 흘러가는 데이터를 포함/ 메모리의 시작주소, 크기, 타임스탬프, 레퍼런스 카운트(얼마나 많은 element가 버퍼를 사용하고 있는지)

  ⇒ 일반적 버퍼 사용은 맨 처음 버퍼가 생성되고 메모리가 할당되며 해당 메모리에 데이터를 복사한다. 복사된 메모리를 가진 버퍼를 다음 element에 넘기고, 다음 element는 버퍼에 쓰여진 데이터를 읽은 후 버퍼의 레퍼런스 카운트를 줄인다.

  ⇒ 버퍼는 항상 source에서 sink로 이동한다.(downstream)
- event
  ⇒ downstream은 데이터 흐름의 동기화에 주로 사용
- message
  pipeline의 메시지 버스 상에서 element에 의해 전송되는 object이다. 주로 error나 tag, 상태 변화, buffer 상태 등의 정보를 전달할 때 사용 element가 application에 thread-safe하게 정보를 전달 할 수 있는 방법이다. (thread-safe - 멀티 스레드 프로그래밍에서 일반적으로 어떤 함수나 변수, 혹은 객체가 여러 스레드로부터 동시에 접근이 이루어져도 프로그램의 실행에 문제가 없음을 뜻함. 보다 엄밀하게는 하나의 함수가 한 스레드로부터 실행 중일 때, 다른 스레드가 그

함수를 호출하여 동시에 함께 실행되더라도 각 스레드에서의 함수의 수행 결과가 올바로 나오는 것으로 정의)
⇒ Thread-safe를 지키기위한 방법
https://gompangs.tistory.com/entry/OS-Thread-Safe%EB%9E%80

- queries
  application 단에서 pipeline에게 재생 위치 등의 정보를 요청할 수 있게 한다.
  queries는 항상 동기적으로 동작하며, element도 연결된 element에게 query를 보내서 정보를 요청할 수 있다.(file size, duration 정도)

```
#include <gst/gst.h>

int main(int argc, char* argv[]){
    GstElement *pipeline;
    GstBus *bus;
    GstMessage *msg;

    gst_init(&argc, &argv);

    pipeline = gst_parse_launch("playbin uri=http://docs.gstreamer.com/media/sintel_trailer-480p.webm", NULL);

    gst_element_set_state(pipeline, GST_STATE_PLAYING);

    bus = gst_element_get_bus(pipeline);
    msg = gst_bus_timed_pop_filtered(bus, GST_CLOCK_TIME_NONE, (GstMessageType)(GST_MESSAGE_ERROR | GST_MESSAGE_EOS));
    if(msg != NULL)
        gst_message_unref(msg);
    gst_object_unref(bus);
    gst_element_set_state(pipeline, GST_STATE_NULL);
    gst_object_unref(pipeline);
    return 0;
}
```

gst_init(&argc, &argv); -> 라이브러리 초기화 / argc, argv로 넘어온 gstreamer 옵션 값을 받아서 실행

gst_parse_launch -> 자동으로 pipeline을 생성
gst_element_set_state -> 파이프 라인 형태의 상태값
->
GST_STATE_NULL, GST_STATE_READY, GST_STATE_PAUSED, GST_STATE_PLAYING

g_signal_connect:

```
g_signal_connect (data.app_source, "need-data", G_CALLBACK (start_feed), &data);
g_signal_connect (data.app_source, "enough-data", G_CALLBACK (stop_feed), &data);
g_signal_connect (data.app_sink, "new-sample", G_CALLBACK (new_sample), &data);
```

g_signal_connect(instance, detailed_signal, c_handler, data)

```
void callback_destroy (GtkWidget *widget, gpointer data  {
  gtk_main_quit ();
}
...
GtkWidget *window;
window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
g_signal_connect (G_OBJECT (window), "destroy", G_CALLBACK (callback_destroy), NULL);
...
```

destroy -> widget이 제거될 때 발생하는 Signal이다.

```
g_signal_connect (app->appsrc, "need-data", G_CALLBACK (start_feed), app);
g_signal_connect (app->appsrc, "enough-data", G_CALLBACK (stop_feed), app);
```

signal에 사용자의 정의 handler인 callback(다른 함수를 호출해 일을 시키고 나는 계속 일을 하고 있는 것?) 함수를 연결해 준다.

```
typedef enum  {
  G_SIGNAL_RUN_FIRST  = 1 << 0,
  G_SIGNAL_RUN_LAST   = 1 << 1,
  G_SIGNAL_RUN_CLEANUP        = 1 << 2,
  G_SIGNAL_NO_RECURSE = 1 << 3,
  G_SIGNAL_DETAILED   = 1 << 4,
  G_SIGNAL_ACTION     = 1 << 5,
  G_SIGNAL_NO_HOOKS   = 1 << 6
} GSignalFlags;
```

g_signal_connect
g_signal_connect_after

둘 다 플래그 호출 하나 after는 항상 default handler보다 나중에 호출된다.