NXtranslate

Peter F. Peterson

NXtranslate
by Peter F. Peterson

`NXtranslate` is an anything to NeXus converter. This is accomplished by using translation files and a plugin style of architecture where `NXtranslate` can read from new formats as plugins become available. This document describes the usage of `NXtranslate` by three types of individuals: the person using existing translation files to create NeXus files, the person creating translation files, and the person writing new Retrievers. All of these concepts are discussed in detail.

Table of Contents

Chapter 1. Overview

`NXtranslate` is designed to be the anything to NeXus converter. To this end it is built in a modular fashion so the types of files that can be read from can vary between different installations. The reason for this is to minimize the size of the executable. In line with this modularity is the author's desire to work with users of `NXtranslate` to add abilities, clarify documentation, and fix bugs.

`NXtranslate` operates by parsing a translation file to create a NeXus file. The translation file contains the organization of the resulting NeXus file, data, and instructions on other how to obtain data using retrievers. This book is organized into chapters with an increasingly sophisticated user in mind. The chapter you are reading is a general overview on how to use `NXtranslate` with an existing installation and existing translation files. Chapter 2 is aimed at writing translation files and chapter 3 will discuss in more detail what retrievers are and how to write them.

Command line arguments

This section will explain the various command line arguments to `NXtranslate`. For all of the examples here the name of the translation file is `test.xml`.

nxtranslate   [--help]   [-o   *outputfile*]   [--hdf4 | --hdf5]   [-D *macro*] {*translationfile*} [--append *outputfile*]

First to get the easy arguments out of the way. Typing just **nxtranslate** will give a usage statement similar to what is above. **nxtranslate** *--help* will print the full help message. Generally speaking this is not what you are interested in.

The minimum argument list for `NXtranslate` to do anything other than print the usage message is to supply a translation file. The cannonical example is

`bash$` **nxtranslate** `test.xml`

This tells `NXtranslate` to parse the file `test.xml` and produce a NeXus file called `test.nxs` using the default base format (base format is discussed below). To change the name of the output file use the "`-o`" switch.

`bash$` **nxtranslate** `test.xml` *-o* `my_file.nxs`

The only difference with the previous example is that the resulting NeXus file is `my_file.nxs`.

The switches "*--hdf4*" and "*--hdf5*" are mutually exclusive and take no arguments. These are used to select the base format for the output file. NeXus is actually written using the Hierarchical Data Format (HDF) which is produced by the National Center for Supercomputer Applications (NCSA). There are two (incompatible) versions of HDF that have widespread use that are commonly referred to as HDF4 and HDF5. Part of the purpose of the NeXus API is to hide the difference between the different bases. In this spirit `NXtranslate` only exposes the bases with these switches. To create two files with the same structure and diffent bases is easy.

`bash$` **nxtranslate** *--hdf4* `test.xml` *-o* `my_hdf4.nxs`
`bash$` **nxtranslate** *--hdf5* `test.xml` *-o* `my_hdf5.nxs`

The last command line argument is the "*-D*" switch. This switch allows for substituting strings in the the translation file for the *NXS:mime_type* , *NXS:source* , and *NXS:location* attributes in the translation file. To get a better understanding of what this means see Chapter 2. For now it is enough to show an example.

```
bash$ nxtranslate test.xml -DFILE1=old_nexus.nxs
```

This example assumes that there is a macro *FILE1* in the translation file. `NXtranslate` will convert the string *FILE1* into `old_nexus.nxs` before creating the resulting NeXus file. This allows for a script to convert an entire directory of files to look like (using python)

```
listing=glob.glob("*.nxs")
for file in listing:
    os.system("nxtranslate converter.xml -DFILE1=%s -o new_%s" % (file,file))
```

This bit of code (plus the proper import statements) would use the translation file `converter.xml` to translate all `*.nxs` in the current working directory.

Chapter 2. The Translation File

The file produced by `NXtranslate` is entirely determined by the contents of the translation file. This chapter discusses the format of a translation file as well as listing "location strings" for the external formats.

Overview

Translation files are written in xml and read using an xml parser. For this reason they must be a valid xml file. [1] This means that the following rules must be adhered to

- Every opening tag must have a corresponding closing tag at the same level. This means that `<entry><data></data></entry>` is allowed while `<entry><data></entry></data>` and `<entry><data></data>` are not.
- Tags and attribute names are case sensitive. Therefore `<entry>` and `<Entry>` are distinct tags. While this can lead to confusion when writing a translation file it is easily avoided in practice.
- Attribute values must be inside single (') or double (") quotes.
- Tags and attribute names cannot start with a number or special character. Another way of saying this is that the name must start with a letter.
- Certain characters will break the parsing of the xml. The characters, and how to create them are `<`(`<`), `>`(`>`), `&`(`&`), `"`(`"`), and `'`(`'`).
- Empty tags, `<data></data>`, can be replaced with a single tag, `<data/>`. This convenience will make more sense during the discussion of translation files when specifying information outside of the file.

There are some other rules to note about the translation file. It is not simply a XML file, there are additional constraints. However, the translation file is not directly validated to follow these constraints, but failing to follow them will result in the program exiting early without creating a NeXus file. Also, NXtranslate is intended to be used to write any file readable by the NeXus API, so the translation file is not validated against definition files. [4] First some definitions used througout this document.

Translation file definitions

napi

  An abbreviation for the NeXus Abstract Program Interface.

node

  A point in the hierarchy, it can either contain other nodes (be a parent with children) or not (a leaf node). Any pair of opening an closing tags represents a single node.

group

  A node that contains other nodes.

field

  A node that does not contain other nodes (a leaf node). In other places in NeXus this is sometimes refered to as a "data" or a "SDS".

retriever

  An object whose job is to retrieve information from a source external to the translation file. Which retriever is created is determined by the value of

`NXS:mime_type` . The retriever is initialized using the value of `NXS:source` . Information is produced by the retriever using the `NXS:location`.

special attribute

An attribute that is interpreted by `NXtranslate` as a command to deal with external information. The special attributes are `NXS:mime_type` , `NXS:source` , `NXS:location`, and `target` .

`NXS:mime_type`

A keyword that denotes what library to use to retrieve information from an external source. It can be a valid mime type.

`NXS:source`

A string denoting what a retriever should use to initialize itself. This is generally a file on the local system for the retriever to open.

`NXS:location`

A string passed to the retriever for it to generate data from. For example, when using the NeXus retriever this is a path to a particular node in the file which will be written out to the resulting NeXus file.

`NAPIlink`

This denotes a node that is a link to another node in the file. It must have a `target` attribute. All other attributes will be ignored

`target`

The attribute denoting what a `NAPIlink` node should be linked to. The syntax for describing location is the same as for the NeXus retriever. If this attribute appears in a node other than `NAPIlink` it will be treated as a normal attribute.

primative type

Any of the following types (ignoring bit-length): `NX_UINT` (unsigned integer), `NX_INT` (signed integer), `NX_FLOAT` (floating point number), `NX_CHAR` (character), `NX_BOOLEAN` (boolean, or true/false), `NX_BINARY` (binary value). At the moment `NX_BOOLEAN` and `NX_BINARY` are not supported by `NXtranslate` and the NeXus API supports only one dimension arrays of `NX_CHAR`.

Now that the definitions have been presented the other constraints of a translation file can be explained.

- The root node in a file will be `<NXroot>`. There will be nothing before or after it, and only one of them. The NXroot can be used to set global values for `NXS:mime_type` and `NXS:source` .

- Only groups can exist directly inside the root. This is a constraint of the NeXus API.

- Every node (except the `NXroot` and `NAPIlink`) needs a `name` and `type`. If the node has a `NXS:location` then the type can be omitted since the retriever will provide it.

- Groups cannot have any attribute other than the special ones. Fields can have any attribute. This reflects a restriction in the NeXus API and does not constrain the contents of resulting NeXus files in any way.

- Groups cannot have any data in them. In other words things similar to `<data type="NXdata">1 2 3 4</data>` are incorrect.

- To specify the dimensions of a field, use square brackets [] affter the type. A single precision floating point array with five elements would have `type="NX_FLOAT32[5]"`. If the field has only one element, or is a character array, the dimensions can be left off. For character arrays, the dimensions are ignored.

- To specify the type of a attribute denote the primative type separated from the value using square brackets. For numeric types only scalars are allowed. If no type is specified it is assumed to be a character array (length is determined automatically).

Simple Translation

While `NXtranslate` is the anything to NeXus translator, it is possible to have everything specfied in the translation file. Example 2-1 shows a translation file where no information will be taken from any other file.

Example 2-1. Simple translation file `test_simple.xml`

```
   <NXroot>
     <NXentry name="entry1">
       <NXnote name="note">
        <author type="NX_CHAR">George User</author>
 5      <type type="NX_CHAR">text/plain</type>
        <data type="NX_CHAR">The data is a simple parabola, f(x)=x^2
        </data>
       </NXnote>
       <NXdata name="parabola_1D">
10      <x type="NX_INT8[11]" axis="NX_INT:1" units="number">
            0 1 2 3  4  5  6  7  8  9  10
        </x>
        <f_x type="NX_INT16[11]" signal="NX_INT:1" units="number">
            0 1 4 9 16 25 36 49 64 81 100
15      </f_x>
       </NXdata>
     </NXentry>
     <NXentry name="entry2">
       <NXnote name="note">
20      <author type="NX_CHAR">George User</author>
        <type type="NX_CHAR">text/plain</type>
        <data type="NX_CHAR">The data is a two dimensional parabola,
                          f(x,y)=x^2+y^2</data>
       </NXnote>
25     <NXdata name="parabola_2D">
        <x type="NX_FLOAT32[4]" axis="NX_INT:1" units="number">
            1.0 4.7 2.3 1.6
        </x>
        <y type="NX_FLOAT32[3]" axis="NX_INT:2" units="number">
30          3.3 6.2 9.2
        </y>
        <f_x_y type="NX_FLOAT64[4,3]" signal="NX_INT:1" axes="x,y" units="number">
            11.89 32.98 16.18
            13.45 39.44 60.53
35          43.73 41.00 85.64
            106.73 89.93 87.20
        </f_x_y>
       </NXdata>
     </NXentry>
40 </NXroot>
```

This example follows all of the rules laid out in the previous section and serves to introduce several of the features of the translation file. First a style note though, in

XML files there is a concept of "ignorable whitespace". These are carriage returns (\n), line feeds (\r), tabs (\t), and spaces. These are ignored (as suggested by the term "ignorable whitespace") and are present to aid those looking at the raw XML to see the node hierarchy.

The main purpose of Example 2-1 is to show how to specify information in a translation file. Line 4 demonstrates the method for strings. Here the *name* is *author* and the *type* is *NX_CHAR*. The length of the character array is determined from the actual string supplied rather than what is specified in the *type* attribute. The value is created by reading in the supplied string, converting tabs, carriage returns, and line feeds into a single space, turning any sections of multiple whitespace into a single space, then chopping off any whitespace at both ends of the string. This allows the person writting the file to add whitespace in strings as needed to make the raw XML easier to read, without changing what is written into the final NeXus file.

Next to look at is how arrays of numbers are specified. Lines 24-27 show both one and two dimensional arrays. The dimension of the array is specified with the type as discussed above. "The thing to notice here is that arrays of numbers are specified as comma delimited lists. The brackets in the list of values are "syntatic sugar". When the values are read in NXtranslate converts them into commas then converts multiple adjacent commas into a single comma. The purpose of this is so translation file authors can more easily see each dimension of the array that they wrote. The brackets can also be removed altogether as seen in line 24."

Translation from NeXus

Next is to show how to use NXtranslate to bring in information from external sources. Example 2-2 demonstrates various features of importing information from external sources, including modifying it before writing.

Example 2-2. Translation from NeXus file `test_nexus.xml`

```
    <NXroot NXS:source="test_simple.nxs" NXS:mime_type="application/x-NeXus">
      <entry_1D NXS:location="/entry1"/>
      <entry_2D type="NXentry">
        <note NXS:location="/entry1/note">
  5         <description type="NX_CHAR">The functional form of the data
                </description>
        </note>
        <parabola_2D type="NXdata">
          <x axis="2" NXS:location="/entry2/parabola_2D/x"/>
  10        <y axis="1" NXS:location="/entry2/parabola_2D/y"/>
          <f_x_y type="NX_FLOAT64[3,4]" axes=""
                NXS:location="/entry2/parabola_2D/f_x_y"/>
        </parabola_2D>
      </entry_2D>
  15 </NXroot>
```

As suggested earlier the root node (line 1) has defined a *NXS:source* and *NXS:mime_type* to use for creating a retriever. Line 2 demonstrates that entire entries can be copied from one file to the next and that the name of a node can be changed. In this case it is from *entry1* to *entry_1D*. Lines 4-7 show how to copy over an entire group and add a new field to it. For finer control of what is added and have the ability to change attributes look at lines 9-12. Line 11 shows how to change the dimensions of the field by using the *type* attribute. Please note that this will not work for character arrays and the total number of array items must remain constant. Also, the type itself cannot be changed (single precision float to double precision float, etc.). Since the dimensions of the *f_x_y* array change it makes sense to change the axes for plotting. This is done in both line 9 and 10 by specifying the attribute and its new value. To add another attribute just specify it similarly. Line 11

demonstrates erasing the *axes* attribute. Specify the attribute with an empty string as the value.

These two examples have shown the way to set up a translation file. You can import information from multiple files by declaring another *NXS:source* and *NXS:mime_type* . There are a couple of things to know about these as well. The default *NXS:mime_type* is "*application/x-NeXus*" so it does not need to be specified. For each *NXS:source* , whatever *NXS:mime_type* was defined in the parent node will be used for the current *NXS:source* . Example 2-3 shows what, in principle, could be done with NXtranslate as more retrievers get written.[5]

Example 2-3. A contrived example

```
   <NXroot>
     <entry1  NXS:source="test_simple.nxs" NXS:location="/entry1">
       <user type="NXuser" NXS:source="127.0.0.1" NXS:mime_type="mySQL">
         <name type="NX_CHAR">George User</name>
 5       <address NXS:location="query(George User):address"/>
         <email NXS:location="query(George User):email"/>
         <phone NXS:location="query(George User):phone"/>
         <picture NXS:source="GeorgeUser.jpg" NXS:mime_type="img/jpeg" NXS:location="a
       </user>
10   </entry1>
     <entry2 NXS:source="test_nexus.nxs" NXS:location="/entry_2D"/>
   </NXroot>
```

Anatomy of Links

The two nodes involved in a link are the source and link. The source is the original version of the information, the link is the copy. There is no way to decipher which is the original and which is the copy without direct comparison of ids using the NeXus api. Links can be either to a group or field. Links to attributes are not supported by the napi . A link to a group and field are both shown in Example 2-4. The first link is to a group whose name was *group1*, while the second link is to a field *array1*.

Example 2-4. Two links

```
<NAPIlink target="/entry/group1"/>
<NAPIlink target="/entry/group1/array1"/>
```

Strings for Translation

The previous section discussed how to write a translation file and several of its features. This section will explain in more detail the strings available for use in a translation file. In principle this section is incomplete because there may exist retrievers that the authors have not been informed of so consider this list incomplete. Also, by nature, the retrievers are quite decouple so the location strings for each retriever can be significantly different from the others.

NeXus

As seen earlier in this chapter the *NXS:mime_type* for NeXus files is *application/x-NeXus*. Similarly the *NXS:location*strings are as simple as possible. NeXus files are organized hierarchically similar to the translation file. A good analogy is to compare it to a file system where the groups are directories and the fields are files. Using this analogy the *NXS:location* strings are absolute paths to the directory or file to be copied. Since there examples of NeXus location strings

in Example 2-2 and Example 2-3 there is only one other thing to mention, the path separator is a forward slash, "/".

Simple ASCII

The `NXS:mime_type` for the simple ASCII retriever is `text/plain`. The functionality of the simple ASCII retriever is limited. This is to emphasize the methodology for building retrievers, rather than build a general purpose one. All of the location strings are integers defining the line number to use. The first line of the file is zero.

SNS Histogram

The `NXS:mime_type` for the SNS histogram retriever is `application/x-SNS-histogram`.

The `NXS:location` is of the general form

```
[...,dim2,dim1][...,dimY,dimX]#{tag_name_1|operator_1}keyword_1{tag_name_2|
operator2}keyword_2...
```

Notice that the `NXS:location` is divided into two parts, declaration and definition, separated by `#`. The declaration describes the dimension of the retrieved data. The definition describes which information the data consists of. Both of these will be described in greated detail below.

The declaration part, `[...,dim2,dim1][...,dimY,dimX]` surrounded by square brackets, contains between the first brackets the size of each dimension of the array to be returned, separated by commas, and between the second set of brackets, the dimensions of the array to read from. The values are specified as positive integers. The current version of the retriever returns an array of the same size as the initial array, no matter the dimensions given between the first set of brackets.

The definition part, `{tag_name_1|operator_1}keyword_1{tag_name_2|operator2}...`, is where selecting the data to be transfered from the SNS histogram file is described. Each part of the definition consists of a `tag_name` and `operator` separated by a vertical slash `"|"`. Multiple definitions can exist in a single `NXS:location` separated by `keyword`s. If the definition is missing, then all of the available data will be retrieved.

The possible values for the `tag_name` are

`pixelID`

> Select using unique pixel identifiers. Applicable for all detectors.

`pixelX`

> Select using column numbers. Applicable for all area detectors

`pixelY`

> Select using row numbers. Applicable for all area detectors

`Tbin`

> Select using time channels. Applicable for all detectors

The `operator` can be of one of two forms

- `loop(start,end,increment)` is used to specify a series of identifiers that runs inclusively from *start* to *end* in steps of *increment*.

- List of identifiers. The identifiers specify which data to include. The identifiers must be separated by commas.

The *keyword* is used to link various declarations together into unions and intersections. Keywords are entirely optional. Keywords that work on two definitions are left associative.

!

    The logical "not" operator. This negates the definition following it. Must be placed just in front of the curly braces it is associated with.

()

    Grouping operation. This can be used to clarify what order multiple keywords are applied. No associative parentheses are allowed within the curly braces.

AND

    The logical "and" operator. This generates the intersection of two definitions. This parameter is case sensitive.

OR

    The logical "or" operator. This generates the union of two definitions. This parameter is case sensitive.

Examples

[150,256,167][304,256,167]#{pixelID|loop(1,38400,1)}

    This retrieves the first 38400 pixel identifiers and put the data into a 150x256x167 array where the 167 dimension changes the fastest. In this example, there are 167 time channels, 256 columns, and 150 rows. The data are coming from a binary file where the data are stored as a 304x256x167 flat array

[50,256,100][304,256,167]#{pixelID|loop(1,12800,1)}AND{Tbin|loop(1,100,1)}

    This retrieves the union of the first 12800 pixel identifiers with the first 100 time channels then places the data into a 50x256x100 array. One must keep in mind that if the array declared is of a different size than the data defined, an error will be generated.

[7,167][304,256,167]#{pixelX|45,53,60,61,62,34500,34501}

    This retrieves a series of columns.

XML retriever

The *NXS:mime_type* for the XML retriever is *text/xml*. The XML retriever is built on top of libxml2's document object model (DOM) parser. Because of this the entire file for information to be retrived from is loaded into memory as a character arrays. The DOM API was chosen to allow for jumping around the source file without needed to

parse its contents multiple times. The location string will be formatted according to the following rules:

- The location string for a field will look like a (unix) path. Each level of the hierarchy is separated by a forward slash, "/".

- To specify the type the value is preceeded using a name separated using a colon, ":". The allowed names are "INT8,INT16,INT32,UINT8,UINT16,UINT32,FLOAT32,FLOAT64". If no name is specified it is (implicitly) a string. Therefore to get "the_answer" as a double precision float the location is "FLOAT64:/numbers/the_answer".

- In the case where the field has a "type" attribute with the value being one of the types above that will be used rather than as a character array. Specifying the type in the location will override what is in the source file.

- Arrays can be specified as part of the type as either an attribute in the XML file or in the location string. To get a six element integer array use the location "/numbers/array" which points to a whitespace delimited list. Multiple dimensions are specified by using a comma delimited list in the square brackets (i.e. "INT16[3,2]:/numbers/array")

- To get an attribute specify it at the end of a path separated by a hash symbol, "#". Therefore to get attr2 as a single precision float the location is "FLOAT32:/numbers#attr2".

This methodolgy does not allow for automatically detecting the type of an imported attribute (it will be read as a string), or differentiating two fields at the same level with the same tag name.

Notes

1. There are many places to find more information about XML

   2. http://www.w3.org/XML/
   is the definitive standard while

   3. http://studio.tellme.com/general/xmlprimer.html
   has a one page overview of what XML is.

2. http://www.w3.org/XML/

3. http://studio.tellme.com/general/xmlprimer.html

4. This decision was made on the basis of performance since it was determined that most of the time a "standard" translation file will be used to convert a large number of files.

5. While retrievers that import information from mySQL and jpeg images would be nice, they do not currently exist.

Chapter 3. Retriever Details

Example 3-1. listing of `retriever.h`

```
class Retriever{
  typedef Ptr<Retriever> RetrieverPtr;

 public:
  /**
   * The factory will call the constructor with a string. The string
   * specifies where to locate the data (e.g. a filename), but
   * interpreting the string is left up to the implementing code.
   */
  //Retriever(const std::string &);

  /**
   * The destructor must be virtual to make sure the right one is
   * called in the end.
   */
  virtual ~Retriever()=0;

  /**
   * This is the method for retrieving data from a file. The whole
   * tree will be written to the new file immediately after being
   * called. Interpreting the string is left up to the implementing
   * code.
   */
  virtual void getData(const std::string &, tree<Node> &)=0;

  /**
   * This method is to be used for debugging purposes only. While the
   * string can be anything, most useful is "[mime_type] source".
   */
  virtual std::string toString() const=0;

  /**
   * Factory method to create new retrievers.
   */
  static RetrieverPtr getInstance(const std::string &, const std::string &);
};
```

Example 3-1 is the listing of the Retriever abstract base class. In addition to these methods there are a couple of assumptions made about classes that implement this interface

Other constraints

1. The copy constructor and assignment operator will not be used. It is suggested that they are made private methods.

2. There is a static const string called MIME_TYPE which will be used to determine if that particular Retriever should be created by the factory. Care must be made to select a unique MIME_TYPE to prevent name clashing.

3. The destructor will properly deallocate all resources allocated in the construtor. Specifically, if a file is opened in the constructor, it should be closed in the destructor.

4. If anything goes wrong during the course of the Retriever's operation, an std::exception will be thrown.

The rest of this chapter describes how to create the body of code, and header, for an example implementation.

The Simple ASCII Retriever as an Example

The simplest retriever is the the one for the *NXS:mime_typetext/plain*. Because of this it makes a good example of how to create your own retriever. The files are located in the `text_plain` subdirectory as `retriever.h` and `retriever.cpp`.

Example 3-2. listing of `text_plain/retriever.h`

```
#include "../retriever.h"
#include <fstream>

// this is not intended to be inherited from
class TextPlainRetriever: public Retriever{
 public:
  TextPlainRetriever(const std::string &);
  ~TextPlainRetriever();
  void getData(const std::string &, tree<Node> &);
  std::string toString() const;
  static const std::string MIME_TYPE;
 private:
  TextPlainRetriever(const TextPlainRetriever&);
  TextPlainRetriever& operator=(const TextPlainRetriever&);
  std::string source;
  int current_line;
  std::ifstream infile;
};
```

Note that none of the methods are virtual, so this is not intended to be derived from directly. That being said, you may want to copy the header and code for your own retriever as a basis of what works. In this example the copy constructor and assignment operator are made private as specified in Other constraints 1. The private data is a filehandle and the name of the file that is open for reading. The file name and *NXS:mime_type* are used in the `toString` to identify it uniquely for debugging as seen in Example 3-3.

Example 3-3. Listing of simple ascii `toString`

```
string TextPlainRetriever::toString() const{
  return "["+MIME_TYPE+"] "+source;
}
```

The first non-trivial function to write is the constructor. The constructor is not very complicated or insightful. The *source* and accounting for where in the file the reading is (*current_line*) are initialized in line 1. Line 3 opens the file, and line 6 confirms that it was opened without error. An exception is thrown if there is a problem to follows Other constraints 4. The constructor is very brief because C++ *fstream* library provides the *ifstream* object that does most of the work.

Example 3-4. Listing of the simple ascii constructor

```
   TextPlainRetriever::TextPlainRetriever(const string &str): source(str),current_line
     // open the file
     infile.open(source.c_str());

5  // check that open was successful
   if(!infile.is_open())
```

```
      throw invalid_argument("Could not open file: "+source);
  }
```

The destructor for the Retriever in Example 3-5 is just as simple simpler since all it has to do is close the file. There were no calls in the constructor (or anywhere else) to `new` or `malloc` so the constructor does not need to call `delete` or `free`.

Example 3-5. Listing of the simple ascii destructor

```
TextPlainRetriever::~TextPlainRetriever(){
  // close the file
  if(infile)
    infile.close();
}
```

Next is the `getData` function which is simple as well. All that `getData` does is grab a line of text from the file and create a node. Lines 3-4 are error checking, and line 7 converts the *location* string into an integer. Line 10 moves to the appropriate place in the file while line 12 gets the string on that line. Since every `getData` must put a *node* into the provided *tree*, line 15 creates a *node* to be filled with data. Lines 18-20 update the generic *node* with the string read in from the source file. Finally line 21 adds the single node to the supplied tree.

Example 3-6. Listing of the simple ascii `getData`

```
   void TextPlainRetriever::getData(const string &location, tree<Node> &tr){
     // check that the argument is not an empty string
     if(location.size()<=0)
       throw invalid_argument("cannot parse empty string");
 5
     // check that the argument is an integer
     int line_num=string_util::str_to_int(location);

     // set stream to the line before
10   skip_to_line(infile,current_line,line_num);
     // read the line and print it to the console
     string text=read_line(infile);

     // create an empty node
15   Node node("empty","empty");

     // put the data in the node
     vector<int> dims;
     dims.push_back(text.size());
20   update_node_from_string(node,text,dims,Node::CHAR);
     tr.insert(tr.begin(),node);
   }
```

Example 3-6 is brief because it leverages existing functionality. The *ifstream* objects does all of the work of getting information out of a file. `skip_to_line` and `read_line` are very short functions that scan to a point in an ascii file and read from a point to the next end-of-line character, respectively. Finally, the function `update_node_from_string` existed in `NXtranslate`already to assist *node* creation while reading the translation file. The interested reader can look at the source of `node_util.cpp` and `text_plain/retriever.cpp` to see the body of the functions.