GDSII/Tutorial -- Tutorial for using GdsParser and GdsCreator
Version 0.2
Copyright © 2004 SoftJin Infotech Pvt Ltd

Last modified: 29-Nov-2004 Mon 15:27

This tutorial explains how to use the Gdsii library to read and write GDSII files. It assumes that you are familiar with the GDSII file format. If you want to understand the library implementation, read the file Overview.

The main classes you will need are *GdsParser* to read the file and *GdsCreator* to write the file. You might find this tutorial easier to read if you first skimmed the following files.

builder.h	*GdsBuilder,* the interface to *GdsParser*
creator.h	*GdsCreator*
parser.h	*GdsParser*

Exceptions

Methods (member functions) of *GdsParser* and *GdsCreator* may throw the following exceptions.

std::bad_alloc	memory allocation failed
std::runtime_error	other unrecoverable error in parsing or writing

Both exceptions have a message you can retrieve using exception::what().

A parser or creator object that throws an exception cannot be used further. That is, all exceptions are unrecoverable. If you catch an exception, do not call any other method of the object. All you can safely do is destroy the object and display the exception message.

Unlike many other libraries, this one **does not use** exceptions to signal errors in the arguments. In particular, when you create a file with *GdsCreator* it is up to you to call the methods in the right order and pass them the right arguments. If you pass invalid arguments to *GdsCreator* methods, anything can happen -- an invalid GDSII file, an assertion failure, or a segmentation fault.

GdsCreator

To use *GdsCreator* you must #include its interface file *creator.h*. Here are the public methods of *GdsCreator*.

gdsVersion()	beginBoundary()
beginLibrary()	beginPath()
endLibrary()	beginSref()
beginStructure()	beginAref()
endStructure()	beginNode()
	beginBox()
	beginText()
	addProperty()
	endElement()

To create a valid GDSII file, you must call these methods in a specific order and with correct arguments.

First construct the creator object. The constructor's arguments are the pathname of the GDSII file and a flag that says whether the output file should be gzipped. Then call gdsVersion() to write the HEADER record, followed by beginLibrary() to write the library header records, up to UNITS. Then write all the structures. Finish by calling endLibrary(), which writes the ENDLIB record and closes the file.

The code will look something like this:

```
GdsCreator  creator("foo/bar.gds", FileHandle::FileTypeAuto);

GdsDate  modTime, accTime;
// ... init modTime and accTime
GdsUnits units(1e-3, 1e-9);
creator.beginLibrary("mylibrary", modTime, accTime, units,
                  GdsLibraryOptions() );
```

```
// ... create all the structures
creator.endLibrary();
```

The second argument to the constructor, FileHandle::FileTypeAuto, tells it to gzip the output file if the filename ends with '.gz'. The filename here, 'foo/bar.gds', does not end with '.gz', so GdsCreator will not gzip the output. The second argument can also be FileTypeNormal (meaning "don't gzip") or FileTypeGzip.

Notice the last argument to *beginLibrary*(). If you look at the function prototypes in *creator.h*, you will see that *beginLibrary*(), *beginStructure*(), and all the element-beginning functions like *beginBoundary*() have a parameter called 'options'. This is for the contents of the optional records for the library, structure, or element. If you don't want to write any optional data, simply pass a newly-constructed object of the appropriate options class, as in the example.

The valid sequence of method calls at the outermost level can be described more concisely by a production in Extended BNF:

```
<file> ::=        gdsVersion()
                  beginLibrary()
                  { <structure> }*
                  endLibrary()
```

The notation { <structure> }* means zero or more occurrences of the sequence of calls to create a structure. The remaining productions are similar:

```
<structure> ::= beginStructure()
                  { <element> }*
                  endStructure()

<element>  ::= <beginElem>
                  { addProperty() }*
                  endElement()

<beginElem> ::= beginBoundary()
                  | beginPath()
                  | beginSref()
                  | beginAref()
                  | beginNode()
                  | beginBox()
                  | beginText()
```

The strings that you pass as arguments or in the options records must be normal NUL terminated C strings. The string lengths must be within the limits set by the GDSII specification, e.g., 44 bytes for font name files. GdsCreator takes care of adding the NULL byte padding to make the record length even.

GdsParser

GdsParser is an event-based parser. Instead of building a tree from the input data it generates events, i.e., makes callbacks to application code, as it recognizes parts of the input file. The callbacks are methods of the class *GdsBuilder*. To parse a GDSII file, derive a class from *GdsBuilder*, override the virtual methods, and pass an instance to *GdsParser::parseFile*().

Your code will look something like this:

```
#include "gdsii/parser.h"
#include "gdsii/builder.h"
using namespace Gdsii;

class MyBuilder : public GdsBuilder {
   // override all virtual methods
   virtual void gdsVersion();
   virtual void beginLibrary();
   // ...
};

void DisplayWarning(const char* msg) {
   printf(stderr, "%s\n", msg);
}

void ProcessFile (const char* path) {
   GdsParser  parser(path, FileHandle::FileTypeAuto, DisplayWarning);
   MyBuilder  builder;
   parser.parseFile(&builder);
   // ...
}
```

The constructor's arguments are the pathname of the input file, a flag to say whether the input file is gzipped, and a warning handler. As with GdsCreator, the value FileHandle::FileTypeAuto means that the input file should be considered to be gzipped if and only if its name ends with ".gz".

The warning handler is a functor whose job is to display any warning messages generated by the parser. Pass Null as the handler if you want to ignore warnings. If you want a non-static class method to handle warnings, you must bind it to an instance. For example, if you have a method

MyBuilder::showWarning (const char* msg)

and you want the parser to invoke builder->showWarning() for each warning, use code like this:

```
#include <functional>
using namespace std;

MyBuilder   builder;
GdsParser parser(path,
             bind1st(mem_fun(&MyBuilder::showWarning),
                   &builder));
```

GdsParser invokes the builder's methods in the same order in which you are expected to invoke *GdsCreator*'s methods. In *MyBuilder*'s methods you can do whatever you want with the data -- build a tree or write out in some other format.

GdsParser::parseFile() parses an entire file. You can also parse a single structure (cell). Call *parseStructure*() and pass it the structure name and builder. You can do this repeatedly, for the same structure or different structures.

Use *GdsParser::makeIndex*() to get a list of the structures in the file and their starting file offsets. This returns a pointer to a *FileIndex* object. The parser owns the index, so do not delete or modify it. Use the index's *getAllNames*() to retrieve all the structure names. For example,

```
#include <algorithm>
#include <vector>

std::vector<const char*>  snames;
FileIndex*  index = parser.makeIndex();
snames.reserve(index->size());
index->getAllNames(std::back_inserter(snames));
```

The starting offset of a structure is useful if you want to convert selected structures in a GDSII file to ASCII. See *ConvertGdsToAscii*() in *asc-conv.cc*.

Sometimes you don't want to parse an entire GDSII file but just want the structure hierarchy: the names of all the structures and an indication of which structure references which in an SREF or AREF. You can build the hierarchy (it's actually a DAG, not a tree) by calling *parseFile*() and overriding just *beginStructure*(), *beginSref*() and *beginAref*() in your builder class. But it is more efficient to use the class *GdsGraphBuilder*, which is similar to *GdsBuilder* but is specialized for this job. It has only four methods to override:

```
beginLibrary()
enterStructure()
addSref()
endLibrary()
```

See *builder.h* for the semantics.

GdsFilter

In this section we see how to use *GdsParser* and *GdsCreator* to solve a simple problem.

Write a program whose input is a GDSII file and a list of layers, and output is a filtered GDSII file containing only the elements in the given layers.

Following the style of the rest of the code, we shall do this by defining a class *GdsFilter* whose constructor takes two pathnames and a vector of layer numbers as its arguments.

```
GdsFilter (const char* infilename,
           const char* outfilename,
           const vector<int>& layers)
```

We can use *GdsParser* to read the input file and *GdsCreator* to write the output file. We also need a subclass of *GdsBuilder* to give *GdsParser::parseFile*(). Let's make that *GdsFilter* itself. It can instantiate a *GdsParser* and *GdsCreator*, passing itself to *GdsParser::parseFile*().

GdsFilter also has to store the selected layer numbers, and for that a *hash_set* is most convenient. So the class declaration will begin like this:

```
class GdsFilter : public GdsBuilder {
    GdsParser     parser;
    GdsCreator    creator;
    hash_set<int> layers;
// other stuff
};
```

GdsParser's constructor requires a *WarningHandler* argument. Because only the code at the user interface will know what to do with warnings, it is not appropriate to handle them in this class. So *GdsFilter*'s constructor must also take a *WarningHandler* argument and pass it to *GdsParser*. The constructor will therefore become something like this:

```
GdsFilter::GdsFilter (const char* infilename,
                      const char* outfilename,
                      const vector<int>& layers,
                      WarningHandler warner)
  : parser(infilename, warner),
    creator(outfilename),
    layers(layers.begin(), layers.end())
{
}
```

As with *GdsToAsciiConverter* and *AsciiToGdsConverter*, the *GdsFilter* constructor simply sets things up. We add another method to do the work:

```
void
GdsFilter::filterFile() {
    parser.parseFile(this);
}
```

Non-element methods

We now have to override *GdsBuilder*'s virtual methods. The following methods have nothing to do with elements or layers, so we just have to forward the call to the creator.

```
gdsVersion()
beginLibrary()
endLibrary()
beginStructure()
endStructure()
```

So, for example, we have,

```
void
GdsFilter::gdsVersion (int version) {
    creator.gdsVersion(version);
}
```

```
void  beginLibrary (const char* libName,
                    const GdsDate& modTime,
                    const GdsDate& accTime,
                    const GdsUnits& units,
                    const GdsLibraryOptions& options)
{
    creator.beginLibrary(libName, modTime, accTime, units, options);
}
```

The remaining methods are similar. If we want to stick to the letter of the GDSII specification and make the output a filtered file rather than an archive file, we have to do some extra work in *beginLibrary*(): set *options.format* to *GDSII_Filtered* and add all the layers to *options.masks.*

Element methods

On to the element-handling methods, where the real work begins. For each element, *parseFile*() calls first the appropriate *beginXXX*() method and then *endElement*(). If the element has properties, *parseFile*() calls *addProperty*() for each property before it calls *endElement*().

The *beginXXX*() method should check the layer number and forward the call to the creator only if it is in the selected set. But *addProperty*() and *endElement*() need to know whether they should call the *GdsCreator* methods. So let us add a bool member *elementSelected* to remember whether the current element is in the selected set.

```
bool  elementSelected;
```

The constructor should initialize this to false. Each *beginXXX*() method for elements sets it to true if the element's layer is in the selected set. For example,

```
void
GdsFilter::beginBoundary (int layer, int datatype,
                          const GdsPointList& points,
                          const GdsElementOptions& options)
{
    if (layers.find(layer) != layer.end()) {
        creator.beginBoundary(layer, datatype, points, options);
        elementSelected = true;
    }
}
```

The other *beginXXX*() methods are similar, except for *beginSref*() and *beginAref*(). Those unconditionally call the corresponding method on creator because they don't belong to a specific layer. *addProperty*() and *endElement*() should call the corresponding method on creator only if *elementSelected* is true.

```
void
GdsFilter::addProperty (int attr, const char* value)
{
   if (elementSelected)
      creator.addProperty(attr, value);
}

void
GdsFilter::endElement()
{
   if (elementSelected)
      creator.endElement();
   elementSelected = false;
}
```

That's about it.

GdsFilter and GdsCreator

Many of the *GdsFilter* methods merely forwarded the call to the creator. You might have wondered why we did not just derive GdsFilter from GdsCreator and override only the element methods involved in filtering. The filtering methods could invoke the base class's method. For example, addProperty() would become

```
void
GdsFilter::addProperty (int attr, const char* value)
{
   if (elementSelected)
      GdsCreator::addProperty(attr, value);
}
```

Yes, this approach is possible, and it is simpler than the one we have presented. It is fine if you don't plan to use GdsFilter outside this one program. But it is logically incorrect to derive *GdsFilter* from *GdsCreator* because a *GdsFilter* is not a *GdsCreator*. You cannot substitute a *GdsFilter* object where a *GdsCreator* is expected.

Deleting empty structures

There is one problem with the solution we presented. If none of the elements in a structure is selected, we write an empty structure to the output. It would be better to delete the structure altogether, and also delete all the SREFs and AREFs that refer to it. A structure in which nothing remains but SREFs and AREFs to empty structures should also be considered empty and deleted.

We cannot do this in one pass of the input file because an SREF may refer to a structure that comes after it. We need two passes. The first pass collects a list of all structures and notes which are empty. The second pass is like the *GdsFilter* code above, except that it also omits empty cells.

Since the two passes do different jobs, we need different builders for each. The first pass makes a structure digraph. Each node in the digraph represents one structure and has a flag that says whether the structure is empty. The node constructor initializes the flag to empty. Each element-handling function in this builder just sets the node flag of the current structure to non-empty if it sees an element whose layer is in the selected set.

beginSref() and *beginAref*() add an edge from the referenced structure's node to the current structure's node, if one does not already exist. So if structure X contains an SREF or AREF to structure Y, the graph will contain an edge from Y to X.

endLibrary() traverses the graph propagating the non-empty status. That is, if some node is non-empty then all nodes reachable from it are also marked non-empty. This ends the first pass of the input. Note that we cannot use buildStructureGraph() for this pass because we also need to note which structures are non-empty.

The second pass, with a different builder, uses the graph built by the first builder. This builder's beginStructure(), endStructure(), beginAref() and beginSref() don't blindly forward the call to the creator. They first check the digraph node for the structure and forward the call only if the node is marked non-empty. The remaining element methods behave like those of GdsFilter above.

The details of this construction are left as an exercise for the reader.