io-funcs.h

Go to the documentation of this file.

```
// base/io-funcs.h

// Copyright 2009-2011  Microsoft Corporation;  Saarland University;
//                      Jan Silovsky;   Yanmin Qian

// See ../../COPYING for clarification regarding multiple authors
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at

//  http://www.apache.org/licenses/LICENSE-2.0

// THIS CODE IS PROVIDED *AS IS* BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
// KIND, EITHER EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED
// WARRANTIES OR CONDITIONS OF TITLE, FITNESS FOR A PARTICULAR PURPOSE,
// MERCHANTABLITY OR NON-INFRINGEMENT.
// See the Apache 2 License for the specific language governing permissions and
// limitations under the License.

#ifndef KALDI_BASE_IO_FUNCS_H_
#define KALDI_BASE_IO_FUNCS_H_

// This header only contains some relatively low-level I/O functions.
// The full Kaldi I/O declarations are in ../util/kaldi-io.h
// and ../util/kaldi-table.h
// They were put in util/ in order to avoid making the Matrix library
// dependent on them.

#include <cctype>
#include <vector>
#include <string>
#include "base/kaldi-common.h"

namespace kaldi {



/*
  This comment describes the Kaldi approach to I/O.  All objects can be written
  and read in two modes: binary and text.  In addition we want to make the I/O
  work if we redefine the typedef "BaseFloat" between floats and doubles.
  We also want to have control over whitespace in text mode without affecting
  the meaning of the file, for pretty-printing purposes.

  Errors are handled by throwing an exception (std::runtime_error).

  For integer and floating-point types (and boolean values):

   WriteBasicType(std::ostream &, bool binary, const T&);
   ReadBasicType(std::istream &, bool binary, T*);

  and we expect these functions to be defined in such a way that they work when
  the type T changes between float and double, so you can read float into double
  and vice versa].  Note that for efficiency and space-saving reasons, the Vector
  and Matrix classes do not use these functions [but they preserve the type
  interchangeability in their own way]

  For a class (or struct) C:
  class C {
  ..
    Write(std::ostream &, bool binary, [possibly extra optional args for specific
classes]) const;
    Read(std::istream &, bool binary, [possibly extra optional args for specific
classes]);
  ..
  }
  NOTE: The only actual optional args we used are the "add" arguments in
```

```
  Vector/Matrix classes, which specify whether we should sum the data already
  in the class with the data being read.

  For types which are typedef's involving stl classes, I/O is as follows:
  typedef std::vector<std::pair<A, B> > MyTypedefName;

  The user should define something like:

   WriteMyTypedefName(std::ostream &, bool binary, const MyTypedefName &t);
   ReadMyTypedefName(std::ostream &, bool binary, MyTypedefName *t);

  The user would have to write these functions.

  For a type std::vector<T>:

   void WriteIntegerVector(std::ostream &os, bool binary, const std::vector<T>
&v);
   void ReadIntegerVector(std::istream &is, bool binary, std::vector<T> *v);

  For other types, e.g. vectors of pairs, the user should create a routine of the
  type WriteMyTypedefName.  This is to avoid introducing confusing templated
functions;
  we could easily create templated functions to handle most of these cases but
they
  would have to share the same name.

  It also often happens that the user needs to write/read special tokens as part
  of a file.  These might be class headers, or separators/identifiers in the
class.
  We provide special functions for manipulating these.  These special tokens must
  be nonempty and must not contain any whitespace.

    void WriteToken(std::ostream &os, bool binary, const char*);
    void WriteToken(std::ostream &os, bool binary, const std::string & token);
    int Peek(std::istream &is, bool binary);
    void ReadToken(std::istream &is, bool binary, std::string *str);
    void PeekToken(std::istream &is, bool binary, std::string *str);


  WriteToken writes the token and one space (whether in binary or text mode).

  Peek returns the first character of the next token, by consuming whitespace
  (in text mode) and then returning the peek() character.  It returns -1 at EOF;
  it doesn't throw.  It's useful if a class can have various forms based on
  typedefs and virtual classes, and wants to know which version to read.

  ReadToken allow the caller to obtain the next token.  PeekToken works just
  like ReadToken, but seeks back to the beginning of the token.  A subsequent
  call to ReadToken will read the same token again.  This is useful when
  different object types are written to the same file; using PeekToken one can
  decide which of the objects to read.

  There is currently no special functionality for writing/reading strings (where
the strings
  contain data rather than "special tokens" that are whitespace-free and
nonempty).  This is
  because Kaldi is structured in such a way that strings don't appear, except as
OpenFst symbol
  table entries (and these have their own format).


  NOTE: you should not call ReadIntegerType and WriteIntegerType with types,
  such as int and size_t, that are machine-independent -- at least not
  if you want your file formats to port between machines.  Use int32 and
  int64 where necessary.  There is no way to detect this using compile-time
  assertions because C++ only keeps track of the internal representation of
  the type.
*/



template<class T> void WriteBasicType(std::ostream &os, bool binary, T t);

template<class T> void ReadBasicType(std::istream &is, bool binary, T *t);

```

```
141
142 // Declare specialization for bool.
143 template<>
144 void WriteBasicType<bool>(std::ostream &os, bool binary, bool b);
145
146 template <>
147 void ReadBasicType<bool>(std::istream &is, bool binary, bool *b);
148
149 // Declare specializations for float and double.
150 template<>
151 void WriteBasicType<float>(std::ostream &os, bool binary, float f);
152
153 template<>
154 void WriteBasicType<double>(std::ostream &os, bool binary, double f);
155
156 template<>
157 void ReadBasicType<float>(std::istream &is, bool binary, float *f);
158
159 template<>
160 void ReadBasicType<double>(std::istream &is, bool binary, double *f);
161
162 // Define ReadBasicType that accepts an "add" parameter to add to
163 // the destination.  Caution: if used in Read functions, be careful
164 // to initialize the parameters concerned to zero in the default
165 // constructor.
166 template<class T>
167 inline void ReadBasicType(std::istream &is, bool binary, T *t, bool add) {
168   if (!add) {
169     ReadBasicType(is, binary, t);
170   } else {
171     T tmp = T(0);
172     ReadBasicType(is, binary, &tmp);
173     *t += tmp;
174   }
175 }
176
178 template<class T> inline void WriteIntegerVector(std::ostream &os, bool binary,
179                                                  const std::vector<T> &v);
180
182 template<class T> inline void ReadIntegerVector(std::istream &is, bool binary,
183                                                 std::vector<T> *v);
184
187 void WriteToken(std::ostream &os, bool binary, const char *token);
188 void WriteToken(std::ostream &os, bool binary, const std::string & token);
189
192 int Peek(std::istream &is, bool binary);
193
195 void ReadToken(std::istream &is, bool binary, std::string *token);
196
201 int PeekToken(std::istream &is, bool binary);
202
205 void ExpectToken(std::istream &is, bool binary, const char *token);
206 void ExpectToken(std::istream &is, bool binary, const std::string & token);
207
211 void ExpectPretty(std::istream &is, bool binary, const char *token);
212 void ExpectPretty(std::istream &is, bool binary, const std::string & token);
213
215
216
220 inline void InitKaldiOutputStream(std::ostream &os, bool binary);
221
225 inline bool InitKaldiInputStream(std::istream &is, bool *binary);
226
227 }  // end namespace kaldi.
228
229 #include "base/io-funcs-inl.h"
230
231 #endif  // KALDI_BASE_IO_FUNCS_H_
```