PBErr

P. Baillehache

March 16, 2021

Contents

Introduction

PBErr is a C library providing structures and functions to manage exception at runtime.

It uses no external library.

1 Interface

```
// ============ PBERR.H ================

#ifndef PBERR_H
#define PBERR_H

// ================= Include =================

#include <stdlib.h>
#include <stdio.h>
```

```c
#include <stdbool.h>
#include <execinfo.h>
#include <errno.h>
#include <string.h>
#include <setjmp.h>
#include <signal.h>


// ================= Define ==================

#define PBERR_MAXSTACKHEIGHT 10
#define PBERR_MSGLENGTHMAX 256

// ================= Data structure ==================

typedef enum PBErrType {
  PBErrTypeUnknown,
  PBErrTypeMallocFailed,
  PBErrTypeNullPointer,
  PBErrTypeInvalidArg,
  PBErrTypeUnitTestFailed,
  PBErrTypeOther,
  PBErrTypeInvalidData,
  PBErrTypeIOError,
  PBErrTypeNotYetImplemented,
  PBErrTypeRuntimeError,
  PBErrTypeNb
} PBErrType;

typedef struct PBErr {
  // Error message
  char _msg[PBERR_MSGLENGTHMAX];
  // Error type
  PBErrType _type;
  // Stream for output
  FILE* _stream;
  // Fatal mode, if true exit when catch
  bool _fatal;
} PBErr;

// ================= Global variable ==================

extern PBErr thePBErr;
extern PBErr* PBMathErr;
extern PBErr* GSetErr;
extern PBErr* ELORankErr;
extern PBErr* ShapoidErr;
extern PBErr* BCurveErr;
extern PBErr* GenBrushErr;
extern PBErr* FracNoiseErr;
extern PBErr* GenAlgErr;
extern PBErr* GradErr;
extern PBErr* KnapSackErr;
extern PBErr* NeuraNetErr;
extern PBErr* PBPhysErr;
extern PBErr* GenTreeErr;
extern PBErr* JSONErr;
extern PBErr* MiniFrameErr;
extern PBErr* PixelToPosEstimatorErr;
extern PBErr* PBDataAnalysisErr;
extern PBErr* PBImgAnalysisErr;
extern PBErr* PBFileSysErr;
```

```
extern PBErr* SDSIAErr;
extern PBErr* GDataSetErr;
extern PBErr* ResPublishErr;
extern PBErr* TheSquidErr;
extern PBErr* CBoErr;
extern PBErr* CrypticErr;
extern PBErr* GradAutomatonErr;
extern PBErr* SmallyErr;
extern PBErr* BuzzyErr;
extern PBErr* NeuraMorphErr;

// ================ Functions declaration ====================

// Static constructor
PBErr PBErrCreateStatic(void);

// Reset thePBErr
void PBErrReset(PBErr* const that);

// Hook for error handling
void PBErrCatch(PBErr* const that);

// Print the PBErr 'that' on 'stream'
void PBErrPrintln(const PBErr* const that, FILE* const stream);

// Secured malloc
#if defined(PBERRALL) || defined(PBERRSAFEMALLOC)
  void* PBErrMalloc(PBErr* const that, const size_t size);
#else
  #define PBErrMalloc(That, Size) malloc(Size)
#endif

// Secured I/O
#if defined(PBERRALL) || defined(PBERRSAFEIO)
  FILE* PBErrOpenStreamIn(PBErr* const that, const char* const path);
  FILE* PBErrOpenStreamOut(PBErr* const that, const char* const path);
  void PBErrCloseStream(PBErr* const that, FILE* const fd);

  bool _PBErrScanfShort(PBErr* const that,
    FILE* const stream, const char* const format, short* const data);
  bool _PBErrScanfInt(PBErr* const that,
    FILE* const stream, const char* const format, int* const data);
  bool _PBErrScanfFloat(PBErr* const that,
    FILE* const stream, const char* const format, float* const data);
  bool _PBErrScanfStr(PBErr* const that,
    FILE* const stream, const char* const format, char* const data);

  bool _PBErrPrintfShort(PBErr* const that,
    FILE* const stream, const char* const format, const short data);
  bool _PBErrPrintfInt(PBErr* const that,
    FILE* const stream, const char* const format, const int data);
  bool _PBErrPrintfLong(PBErr* const that,
    FILE* const stream, const char* const format, const long data);
  bool _PBErrPrintfFloat(PBErr* const that,
    FILE* const stream, const char* const format, const float data);
  bool _PBErrPrintfStr(PBErr* const that,
    FILE* const stream, const char* const format,
    const char* const data);
#else
  #define PBErrOpenStreamIn(Err, Path) \
    fopen(Path, "r")
  #define PBErrOpenStreamOut(Err, Path) \
```

```
    fopen(Path, "w")
  #define PBErrCloseStream(Err, Stream) \
    fclose(Stream)

  #define PBErrScanf(Err, Stream, Format, Data) \
    (fscanf(Stream, Format, Data) == EOF)
  #define PBErrPrintf(Err, Stream, Format, Data) \
    (fprintf(Stream, Format, Data) < 0)
#endif

// Hook for invalid polymorphisms
void PBErrInvalidPolymorphism(void*t, ...);

// Try/catch

// List of exceptions ID, must starts at 1 (0 is reserved for the setjmp at
// the beginning of the TryCatch blocks). One can extend the list at will
// here, or user-defined exceptions can be added directly in the user code
// as follows:
// enum UserDefinedExceptions {
//
//   myUserExceptionA = TryCatchException_LastID,
//   myUserExceptionB,
//   myUserExceptionC
//
// };
// TryCatchException_LastID is not an exception but a convenience to
// create new exceptions (as in the example above) while ensuring
// their ID doesn't collide with the ID of exceptions in TryCatchException.
// Exception defined here are only examples, one should create a list of
// default exceptions according to the planned use of this trycatch module.
enum TryCatchException {

  TryCatchException_test = 1,
  TryCatchException_NaN,
  TryCatchException_Segv,
  TryCatchException_LastID

};

// Function called at the beginning of a TryCatch block to guard against
// overflow of the stack of jump_buf
void TryCatchGuardOverflow(
  // No parameters
  void);

// Function called to get the jmp_buf on the top of the stack when
// starting a new TryCatch block
jmp_buf* TryCatchGetJmpBufOnStackTop(
  // No parameters
  void);

// Function called when a raised TryCatchException has not been caught
// by a Catch segment
void TryCatchDefault(
  // File where the exception occured
  char const* const filename,
  // Line where the exception occured
          int const line);

// Function called at the end of a TryCatch block
void TryCatchEnd(
```

```
  // No parameters
  void);

// Head of the TryCatch block, to be used as
//
// Try {
//   /*... code of the TryCatch block here ...*/
//
// Comments on the macro:
//   // Guard against recursive incursion overflow
//   TryCatchGuardOverflow();
//   // Memorise the jmp_buf on the top of the stack, setjmp returns 0
//   switch (setjmp(*TryCatchGetJmpBufOnStackTop())) {
//     // Entry point for the code of the TryCatch block
//     case 0:
#define Try \
  TryCatchGuardOverflow(); \
  switch (setjmp(*TryCatchGetJmpBufOnStackTop())) { \
    case 0:

// Catch segment in the TryCatch block, to be used as
//
// Catch (/*... one of TryCatchException or user-defined exception ...*/) {
//   /*... code executed if the exception has been raised in the
//     TryCatch block ...*/
//
// Comments on the macro:
//      // End of the previous case
//      break;
//    // case of the raised exception
//    case e:
#define Catch(e) \
      break;\
    case e:

// Macro to assign several exceptions to one Catch segment in the TryCatch
// block, to be used as
//
// Catch (/*... one of TryCatchException or user-defined exception ...*/)
// CatchAlso (/*... another one ...*/) {
// /*... as many CatchAlso statement as your need ...*/
//   /*... code executed if one of the exception has been raised in the
//     TryCatch block ...
//     (Use TryCatchGetLastExc() if you need to know which exception as
//     been raised) */
//
// Comments on the macro:
//    // case of the raised exception
//    case e:
#define CatchAlso(e) \
    case e:

// Macro to declare the default Catch segment in the TryCatch
// block, must be the last Catch segment in the TryCatch block,
// to be used as
//
// CatchDefault {
//   /*... code executed if an exception has been raised in the
//     TryCatch block and hasn't been catched by a previous Catch segment...
//     (Use TryCatchGetLastExc() if you need to know which exception as
//     been raised) */
//
```

```
// Comments on the macro:
//    // default case
//    default:
#define CatchDefault \
      break;\
    default:

// Tail of the TryCatch block if it doesn't contain CatchDefault,
// to be used as
//
// } EndTry;
//
// Comments on the macro:
//      // End of the previous case
//      break;
//    // default case, i.e. any raised exception which hasn't been caught
//    // by a previous Catch is caught here
//    default:
//      // Processing of uncaught exception
//      TryCatchDefault();
//  // End of the switch statement at the head of the TryCatch block
//  }
//  // Post processing of the TryCatchBlock
//  TryCatchEnd()
#define EndTry \
      break; \
    default: \
      TryCatchDefault(__FILE__, __LINE__); \
  } \
  TryCatchEnd()

// Tail of the TryCatch block if it contains CatchDefault,
// to be used as
//
// } EndTryWithDefault;
//
// Comments on the macro:
//  // End of the switch statement at the head of the TryCatch block
//  }
//  // Post processing of the TryCatchBlock
//  TryCatchEnd()
#define EndTryWithDefault \
  } \
  TryCatchEnd()

// Function called to raise the TryCatchException 'exc'
void Raise(
  // The TryCatchException to raise. Do not use the type enum
  // TryCatchException to allow the user to extend the list of exceptions
  // with user-defined exception outside of enum TryCatchException.
  int exc);

// The struct siginfo_t used to handle the SIGSEV is not defined in
// ANSI C, guard against this.
#ifndef __STRICT_ANSI__

// Function to set the handler function of the signal SIGSEV and raise
// TryCatchException_Segv upon reception of this signal. Must have been
// called before using Catch(TryCatchException_Segv)
void TryCatchInitHandlerSigSegv(
  // No parameters
  void);
```

```
#endif

// Function to get the ID of the last raised exception
int TryCatchGetLastExc(
  // No parameters
  void);

// Function to convert from enum TryCatchException to char*
char const* TryCatchExceptionToStr(
  // The exception ID
  enum TryCatchException exc);

// ================ Polymorphism ==================

#if defined(PBERRALL) || defined(PBERRSAFEIO)
  #define PBErrScanf(Err, Stream, Format, Data) _Generic(Data, \
    short*: _PBErrScanfShort, \
    int*: _PBErrScanfInt, \
    float*: _PBErrScanfFloat, \
    char*: _PBErrScanfStr, \
    default: PBErrInvalidPolymorphism) (Err, Stream, Format, Data)

  #define PBErrPrintf(Err, Stream, Format, Data) _Generic(Data, \
    short: _PBErrPrintfShort, \
    int: _PBErrPrintfInt, \
    long: _PBErrPrintfLong, \
    float: _PBErrPrintfFloat, \
    char*: _PBErrPrintfStr, \
    default: PBErrInvalidPolymorphism) (Err, Stream, Format, Data)
#endif

#endif
```

2 Code

2.1 pberr.c

```
// ============ PBERR.C ================

// ================ Include =================

#include "pberr.h"

// ================ Define ==================

// Default PBErr
PBErr thePBErr = {._msg[0] = '\0', ._type = PBErrTypeUnknown,
  ._stream = NULL, ._fatal = true};
// Declare a pointer for each repository, by default they are
// all pointing toward the default PBErr, but it allows the
// user to manage separately the errors if necessary
PBErr* PBMathErr = &thePBErr;
PBErr* GSetErr = &thePBErr;
PBErr* ELORankErr = &thePBErr;
PBErr* ShapoidErr = &thePBErr;
PBErr* BCurveErr = &thePBErr;
PBErr* GenBrushErr = &thePBErr;
```

```
PBErr* FracNoiseErr = &thePBErr;
PBErr* GenAlgErr = &thePBErr;
PBErr* GradErr = &thePBErr;
PBErr* KnapSackErr = &thePBErr;
PBErr* NeuraNetErr = &thePBErr;
PBErr* PBPhysErr = &thePBErr;
PBErr* GenTreeErr = &thePBErr;
PBErr* JSONErr = &thePBErr;
PBErr* MiniFrameErr = &thePBErr;
PBErr* PixelToPosEstimatorErr = &thePBErr;
PBErr* PBDataAnalysisErr = &thePBErr;
PBErr* PBImgAnalysisErr = &thePBErr;
PBErr* PBFileSysErr = &thePBErr;
PBErr* SDSIAErr = &thePBErr;
PBErr* GDataSetErr = &thePBErr;
PBErr* ResPublishErr = &thePBErr;
PBErr* TheSquidErr = &thePBErr;
PBErr* CBoErr = &thePBErr;
PBErr* CrypticErr = &thePBErr;
PBErr* GradAutomatonErr = &thePBErr;
PBErr* SmallyErr = &thePBErr;
PBErr* BuzzyErr = &thePBErr;
PBErr* NeuraMorphErr = &thePBErr;

const char* PBErrTypeLbl[PBErrTypeNb] = {
  "unknown",
  "malloc failed",
  "null pointer",
  "invalid arguments",
  "unit test failed",
  "other",
  "invalid data",
  "I/O error",
  "not yet implemented",
  "runtime error"
};

// ================ Functions implementation ====================

// Static constructor
PBErr PBErrCreateStatic(void) {
  PBErr that = {._msg[0] = '\0', ._type = PBErrTypeUnknown,
    ._stream = NULL, ._fatal = true};
  return that;
}

// Reset thePBErr
void PBErrReset(PBErr* const that) {
  if (that == NULL)
    return;
  that->_msg[0] = '\0';
  that->_type = PBErrTypeUnknown;
  that->_fatal = true;
}

// Hook for error handling
// Print the error type, the error message, the stack
// Exit if _fatal == true
// Reset the PBErr
void PBErrCatch(PBErr* const that) {
  if (that == NULL)
    return;
```

```
  FILE* stream = (that->_stream ? that->_stream : stderr);
  fprintf(stream, "---- PBErrCatch ----\n");
  PBErrPrintln(that, stream);
  fprintf(stream, "Stack:\n");
  void* stack[PBERR_MAXSTACKHEIGHT] = {NULL};
  int stackHeight = backtrace(stack, PBERR_MAXSTACKHEIGHT);
  backtrace_symbols_fd(stack, stackHeight, fileno(stream));
  if (errno != 0) {
    fprintf(stream, "errno: %s\n", strerror(errno));
    errno = 0;
  }
  if (that->_fatal) {
    fprintf(stream, "Exiting\n");
    fprintf(stream, "--------------------\n");
    exit(that->_type);
  }
  fprintf(stream, "--------------------\n");
  PBErrReset(that);
}

// Print the PBErr 'that' on 'stream'
void PBErrPrintln(const PBErr* const that, FILE* const stream) {
  // If the PBErr or stream is null
  if (that == NULL || stream == NULL)
    // Nothing to do
    return;
  if (that->_type > 0 && that->_type < PBErrTypeNb)
    fprintf(stream, "PBErrType: %s\n", PBErrTypeLbl[that->_type]);
  if (that->_msg[0] != '\0')
    fprintf(stream, "PBErrMsg: %s\n", that->_msg);
  if (that->_fatal)
    fprintf(stream, "PBErrFatal: true\n");
  else
    fprintf(stream, "PBErrFatal: false\n");
}

// Secured malloc
#if defined(PBERRALL) || defined(PBERRSAFEMALLOC)
void* PBErrMalloc(PBErr* const that, const size_t size) {
  void* ret = malloc(size);
  if (ret == NULL) {
    that->_type = PBErrTypeMallocFailed;
    sprintf(that->_msg, "malloc of %ld bytes failed\n",
      (unsigned long int)size);
    that->_fatal = true;
    PBErrCatch(that);
  }
  return ret;
}
#endif

// Secured I/O
#if defined(PBERRALL) || defined(PBERRSAFEIO)

FILE* PBErrOpenStreamIn(PBErr* const that, const char* const path) {
#if BUILDMODE == 0
  if (that == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'that' is null");
    that->_fatal = true;
    PBErrCatch(that);
  }
```

```
  if (path == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'path' is null");
    that->_fatal = true;
    PBErrCatch(that);
  }
#endif
  FILE* fd = fopen(path, "r");
  if (fd == NULL) {
    that->_type = PBErrTypeIOError;
    sprintf(that->_msg, "fopen failed for %s", path);
    that->_fatal = false;
    PBErrCatch(that);
  }
  return fd;
}

FILE* PBErrOpenStreamOut(PBErr* const that, const char* const path) {
#if BUILDMODE == 0
  if (that == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'that' is null");
    that->_fatal = true;
    PBErrCatch(that);
  }
  if (path == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'path' is null");
    that->_fatal = true;
    PBErrCatch(that);
  }
#endif
  FILE* fd = fopen(path, "w");
  if (fd == NULL) {
    that->_type = PBErrTypeIOError;
    sprintf(that->_msg, "fopen failed for %s", path);
    that->_fatal = false;
    PBErrCatch(that);
  }
  return fd;
}

void PBErrCloseStream(PBErr* const that, FILE* const fd) {
#if BUILDMODE == 0
  if (that == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'that' is null");
    that->_fatal = true;
    PBErrCatch(that);
  }
  if (fd == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'fd' is null");
    that->_fatal = true;
    PBErrCatch(that);
  }
#endif
  (void)that;
  fclose(fd);
}
```

```
bool _PBErrScanfShort(PBErr* const that,
  FILE* const stream, const char* const format, short* const data) {
#if BUILDMODE == 0
  if (that == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'that' is null\n");
    that->_fatal = true;
    PBErrCatch(that);
  }
  if (stream == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'stream' is null\n");
    that->_fatal = true;
    PBErrCatch(that);
  }
  if (format == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'format' is null\n");
    that->_fatal = true;
    PBErrCatch(that);
  }
  if (data == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'data' is null\n");
    that->_fatal = true;
    PBErrCatch(that);
  }
#endif
  // Read from the stream
  if (fscanf(stream, format, data) == EOF) {
    that->_type = PBErrTypeIOError;
    sprintf(that->_msg, "fscanf failed\n");
    that->_fatal = false;
    PBErrCatch(that);
    return false;
  }
  return true;
}

bool _PBErrScanfInt(PBErr* const that,
  FILE* const stream, const char* const format, int* const data) {
#if BUILDMODE == 0
  if (that == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'that' is null\n");
    that->_fatal = true;
    PBErrCatch(that);
  }
  if (stream == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'stream' is null\n");
    that->_fatal = true;
    PBErrCatch(that);
  }
  if (format == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'format' is null\n");
    that->_fatal = true;
    PBErrCatch(that);
  }
  if (data == NULL) {
    that->_type = PBErrTypeNullPointer;
```

```
    sprintf(that->_msg, "'data' is null\n");
    that->_fatal = true;
    PBErrCatch(that);
  }
#endif
  // Read from the stream
  if (fscanf(stream, format, data) == EOF) {
    that->_type = PBErrTypeIOError;
    sprintf(that->_msg, "fscanf failed\n");
    that->_fatal = false;
    PBErrCatch(that);
    return false;
  }
  return true;
}

bool _PBErrScanfFloat(PBErr* const that,
  FILE* const stream, const char* const format, float* const data) {
#if BUILDMODE == 0
  if (that == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'that' is null\n");
    that->_fatal = true;
    PBErrCatch(that);
  }
  if (stream == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'stream' is null\n");
    that->_fatal = true;
    PBErrCatch(that);
  }
  if (format == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'format' is null\n");
    that->_fatal = true;
    PBErrCatch(that);
  }
  if (data == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'data' is null\n");
    that->_fatal = true;
    PBErrCatch(that);
  }
#endif
  // Read from the stream
  if (fscanf(stream, format, data) == EOF) {
    that->_type = PBErrTypeIOError;
    sprintf(that->_msg, "fscanf failed\n");
    that->_fatal = false;
    PBErrCatch(that);
    return false;
  }
  return true;
}

bool _PBErrScanfStr(PBErr* const that,
  FILE* const stream, const char* const format, char* const data) {
#if BUILDMODE == 0
  if (that == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'that' is null\n");
    that->_fatal = true;
```

```
    PBErrCatch(that);
  }
  if (stream == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'stream' is null\n");
    that->_fatal = true;
    PBErrCatch(that);
  }
  if (format == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'format' is null\n");
    that->_fatal = true;
    PBErrCatch(that);
  }
  if (data == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'data' is null\n");
    that->_fatal = true;
    PBErrCatch(that);
  }
#endif
  // Read from the stream
  if (fscanf(stream, format, data) == EOF) {
    that->_type = PBErrTypeIOError;
    sprintf(that->_msg, "fscanf failed\n");
    that->_fatal = false;
    PBErrCatch(that);
    return false;
  }
  return true;
}

bool _PBErrPrintfShort(PBErr* const that,
  FILE* const stream, const char* const format, const short data) {
#if BUILDMODE == 0
  if (that == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'that' is null\n");
    that->_fatal = true;
    PBErrCatch(that);
  }
  if (stream == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'stream' is null\n");
    that->_fatal = true;
    PBErrCatch(that);
  }
  if (format == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'format' is null\n");
    that->_fatal = true;
    PBErrCatch(that);
  }
#endif
  // Print to the stream
  if (fprintf(stream, format, data) < 0) {
    that->_type = PBErrTypeIOError;
    sprintf(that->_msg, "fprintf failed\n");
    that->_fatal = false;
    PBErrCatch(that);
    return false;
  }
```

```
  return true;
}

bool _PBErrPrintfLong(PBErr* const that,
  FILE* const stream, const char* const format, const long data) {
#if BUILDMODE == 0
  if (that == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'that' is null\n");
    that->_fatal = true;
    PBErrCatch(that);
  }
  if (stream == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'stream' is null\n");
    that->_fatal = true;
    PBErrCatch(that);
  }
  if (format == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'format' is null\n");
    that->_fatal = true;
    PBErrCatch(that);
  }
#endif
  // Print to the stream
  if (fprintf(stream, format, data) < 0) {
    that->_type = PBErrTypeIOError;
    sprintf(that->_msg, "fprintf failed\n");
    that->_fatal = false;
    PBErrCatch(that);
    return false;
  }
  return true;
}

bool _PBErrPrintfInt(PBErr* const that,
  FILE* const stream, const char* const format, const int data) {
#if BUILDMODE == 0
  if (that == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'that' is null\n");
    that->_fatal = true;
    PBErrCatch(that);
  }
  if (stream == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'stream' is null\n");
    that->_fatal = true;
    PBErrCatch(that);
  }
  if (format == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'format' is null\n");
    that->_fatal = true;
    PBErrCatch(that);
  }
#endif
  // Print to the stream
  if (fprintf(stream, format, data) < 0) {
    that->_type = PBErrTypeIOError;
    sprintf(that->_msg, "fprintf failed\n");
```

```
    that->_fatal = false;
    PBErrCatch(that);
    return false;
  }
  return true;
}

bool _PBErrPrintfFloat(PBErr* const that,
  FILE* const stream, const char* const format, const float data) {
#if BUILDMODE == 0
  if (that == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'that' is null\n");
    that->_fatal = true;
    PBErrCatch(that);
  }
  if (stream == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'stream' is null\n");
    that->_fatal = true;
    PBErrCatch(that);
  }
  if (format == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'format' is null\n");
    that->_fatal = true;
    PBErrCatch(that);
  }
#endif
  // Print to the stream
  if (fprintf(stream, format, data) < 0) {
    that->_type = PBErrTypeIOError;
    sprintf(that->_msg, "fprintf failed\n");
    that->_fatal = false;
    PBErrCatch(that);
    return false;
  }
  return true;
}

bool _PBErrPrintfStr(PBErr* const that,
  FILE* const stream, const char* const format,
  const char* const data) {
#if BUILDMODE == 0
  if (that == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'that' is null\n");
    that->_fatal = true;
    PBErrCatch(that);
  }
  if (stream == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'stream' is null\n");
    that->_fatal = true;
    PBErrCatch(that);
  }
  if (format == NULL) {
    that->_type = PBErrTypeNullPointer;
    sprintf(that->_msg, "'format' is null\n");
    that->_fatal = true;
    PBErrCatch(that);
  }
```

```
#endif
  // Print to the stream
  if (fprintf(stream, format, data) < 0) {
    that->_type = PBErrTypeIOError;
    sprintf(that->_msg, "fprintf failed\n");
    that->_fatal = false;
    PBErrCatch(that);
    return false;
  }
  return true;
}

#endif

// Try/catch

// Size of the stack of TryCatch blocks, define how many recursive incursion
// of TryCatch blocks can be done, overflow is checked at the beginning of
// each TryCatch blocks with TryCatchGuardOverflow()
// (Set delibarately low here to be able to test it in the example main.c)
#define TryCatchMaxExcLvl 3

// Stack of jmp_buf to memorise the TryCatch blocks
// To avoid exposing this variable to the user, implement any code using
// it as functions here instead of in the #define-s of trycatch.h
jmp_buf tryCatchExcJmp[TryCatchMaxExcLvl];

// Index of the next TryCatch block in the stack of jmp_buf
// To avoid exposing this variable to the user, implement any code using
// it as functions here instead of in the #define-s of trycatch.h
int tryCatchExcLvl = 0;

// ID of the last raised exception
// To avoid exposing this variable to the user, implement any code using
// it as functions here instead of in the #define-s of trycatch.h
// Do not use the type enum TryCatchException to allow the user to extend
// the list of exceptions with user-defined exceptions outside of enum
// TryCatchException.
int tryCatchExc = 0;

// Label of exceptions, must match the declaration of enum TryCatchException
// Take care of index 0 which is unused in the enum
char* TryCatchExceptionStr[TryCatchException_LastID] = {
  "",
  "TryCatchException_test",
  "TryCatchException_NaN",
  "TryCatchException_Segv",
};

// Function called at the beginning of a TryCatch block to guard against
// overflow of the stack of jump_buf
void TryCatchGuardOverflow(
  // No parameters
  void) {

  // If the max level of incursion is reached
  if (tryCatchExcLvl == TryCatchMaxExcLvl) {

    // Print a message on the standard error output and exit
    fprintf(
      stderr,
      "TryCatch blocks recursive incursion overflow, exiting. "
```

```
      "(You can try to raise the value of TryCatchMaxExcLvl in trycatch.c, "
      "it was: %d)\n",
      TryCatchMaxExcLvl);
    exit(EXIT_FAILURE);

  }

}

// Function called to get the jmp_buf on the top of the stack when
// starting a new TryCatch block
jmp_buf* TryCatchGetJmpBufOnStackTop(
  // No parameters
  void) {

  // Reset the last raised exception
  tryCatchExc = 0;

  // Memorise the current jmp_buf at the top of the stack
  jmp_buf* topStack = tryCatchExcJmp + tryCatchExcLvl;

  // Move the index of the top of the stack of jmp_buf to the upper level
  tryCatchExcLvl++;

  // Return the jmp_buf previously at the top of the stack
  return topStack;

}

// Function called to raise the TryCatchException 'exc'
void Raise(
  // The TryCatchException to raise. Do not use the type enum
  // TryCatchException to allow the user to extend the list of exceptions
  // with user-defined exception outside of enum TryCatchException.
  int exc) {

  // If we are in a TryCatch block
  if (tryCatchExcLvl > 0) {

    // Memorise the last raised exception to be able to handle it if
    // it reaches the default case in the swith statement of the TryCatch
    // block
    tryCatchExc = exc;

    // Move to the top of the stack of jmp_buf to the lower level
    tryCatchExcLvl--;

    // Call longjmp with the appropriate jmp_buf in the stack and the
    // raised TryCatchException.
    longjmp(
      tryCatchExcJmp[tryCatchExcLvl],
      exc);

  // Else we are not in a TryCatch block
  } else {

    // Print a message on the standard error stream and ignore the
    // exception
    fprintf(
      stderr,
      "Unhandled exception (%s).\n",
      TryCatchExceptionToStr(exc));
```

```
  }

}

// Function called when a raised TryCatchException has not been caught
// by a Catch segment
void TryCatchDefault(
  // File where the exception occured
  char const* const filename,
  // Line where the exception occured
          int const line) {

  // If we are outside of a TryCatch block
  if (tryCatchExcLvl == 0) {

    // The exception has not been caught by a Catch segment,
    // print a message on the standard error stream and ignore it
    fprintf(
      stderr,
      "Unhandled exception (%s) in %s, line %d.\n",
      TryCatchExceptionToStr(tryCatchExc),
      filename,
      line);

  // Else, the exception has not been caught in the current
  // TryCatch block but may be catchable at lower level
  } else {

    // Move to the lower level in the stack of jmp_buf and raise the
    // exception again
    Raise(tryCatchExc);

  }

}

// Function called at the end of a TryCatch block
void TryCatchEnd(
  // No parameters
  void) {

  // The execution has reached the end of the current TryCatch block,
  // move back to the lower level in the stack of jmp_buf
  if (tryCatchExcLvl > 0) tryCatchExcLvl--;

}

// The struct siginfo_t used to handle the SIGSEV is not defined in
// ANSI C, guard against this.
#ifndef __STRICT_ANSI__

// Handler function to raise the exception TryCatchException_Segv when
// receiving the signal SIGSEV.
void TryCatchSigSegvHandler(
  // Received signal, will always be SIGSEV, unused
  int signal,
  // Info about the signal, unused
  siginfo_t *si,
  // Optional arguments, unused
  void *arg) {
```

```
  // Unused parameters
  (void)signal; (void)si; (void)arg;

  // Raise the exception
  Raise(TryCatchException_Segv);

}

// Function to set the handler function of the signal SIGSEV and raise
// TryCatchException_Segv upon reception of this signal. Must have been
// called before using Catch(TryCatchException_Segv)
void TryCatchInitHandlerSigSegv(
  // No parameters
  void) {

  // Create a struct sigaction to set the handler
  struct sigaction sigActionSegv;
  memset(
    &sigActionSegv,
    0,
    sizeof(struct sigaction));
  sigemptyset(&(sigActionSegv.sa_mask));
  sigActionSegv.sa_sigaction = TryCatchSigSegvHandler;
  sigActionSegv.sa_flags = SA_SIGINFO;

  // Set the handler
  sigaction(
    SIGSEGV,
    &sigActionSegv,
    NULL);

}

// Function to get the ID of the last raised exception
int TryCatchGetLastExc(
  // No parameters
  void) {

  // Return the ID
  return tryCatchExc;

}

// Function to convert from enum TryCatchException to char*
char const* TryCatchExceptionToStr(
  // The exception ID
  enum TryCatchException exc) {

  return TryCatchExceptionStr[exc];

}

#endif
```

3 Makefile

```
# Build mode
```

```
# 0: development (max safety, no optimisation)
# 1: release (min safety, optimisation)
# 2: fast and furious (no safety, optimisation)
BUILD_MODE?=1

all: pbmake_wget main

# Automatic installation of the repository PBMake in the parent folder
pbmake_wget:
if [ ! -d ../PBMake ]; then wget https://github.com/BayashiPascal/PBMake/archive/master.zip; unzip master.zip; rm -f

# Makefile definitions
MAKEFILE_INC=../PBMake/Makefile.inc
include $(MAKEFILE_INC)

# Rules to make the executable
repo=pberr
$($(repo)_EXENAME): \
$($(repo)_EXENAME).o \
$($(repo)_EXE_DEP) \
$($(repo)_DEP)
$(COMPILER) `echo "$($(repo)_EXE_DEP) $($(repo)_EXENAME).o" | tr ' ' '\n' | sort -u` $(LINK_ARG) $($(repo)_LINK_ARG)

$($(repo)_EXENAME).o: \
$($(repo)_DIR)/$($(repo)_EXENAME).c \
$($(repo)_INC_H_EXE) \
$($(repo)_EXE_DEP)
$(COMPILER) $(BUILD_ARG) $($(repo)_BUILD_ARG) `echo "$($(repo)_INC_DIR)" | tr ' ' '\n' | sort -u` -c $($(repo)_DIR)/$
```

4 Unit tests

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <math.h>
#include "pberr.h"

void UnitTestCreateStatic() {
  printf("UnitTestCreateStatic\n");
  PBErr err = PBErrCreateStatic();
  PBErrPrintln(&err, stdout);
}

void UnitTestReset() {
  printf("UnitTestReset\n");
  PBErr err = PBErrCreateStatic();
  PBErr clone = err;
  memset(&err, 0, sizeof(PBErr));
  PBErrReset(&err);
    printf("Reset ");
  if (memcmp(&err, &clone, sizeof(PBErr)) == 0)
    printf("OK");
  else
    printf("NOK");
  printf("\n");
}
```

```
void UnitTestMalloc() {
  printf("UnitTestMalloc\n");
  char* arr = PBErrMalloc(&thePBErr, 2);
  printf("Malloc ");
  if (arr == NULL)
    printf("NOK");
  else
    printf("OK");
  printf("\n");
  arr[0] = 0;
  arr[1] = 1;
  free(arr);
}

void UnitTestIO() {
  FILE* fd = PBErrOpenStreamOut(&thePBErr, "./testio.txt");
  short a = 1;
  PBErrPrintf(&thePBErr, fd, "%hi\n", a);
  short b = 2;
  PBErrPrintf(&thePBErr, fd, "%i\n", b);
  float c = 3.0;
  PBErrPrintf(&thePBErr, fd, "%f\n", c);
  char* d = "string";
  PBErrPrintf(&thePBErr, fd, "%s\n", d);
  PBErrCloseStream(&thePBErr, fd);
  fd = PBErrOpenStreamIn(&thePBErr, "./testio.txt");
  short checka;
  PBErrScanf(&thePBErr, fd, "%hi", &checka);
  if (a != checka) {
    thePBErr._stream = stdout;
    thePBErr._type = PBErrTypeUnitTestFailed;
    sprintf(thePBErr._msg, "UnitTestIO failed");
    thePBErr._fatal = false;
    PBErrCatch(&thePBErr);
  }
  int checkb;
  PBErrScanf(&thePBErr, fd, "%i", &checkb);
  if (b != checkb) {
    thePBErr._stream = stdout;
    thePBErr._type = PBErrTypeUnitTestFailed;
    sprintf(thePBErr._msg, "UnitTestIO failed");
    thePBErr._fatal = false;
    PBErrCatch(&thePBErr);
  }
  float checkc;
  PBErrScanf(&thePBErr, fd, "%f", &checkc);
  if (fabs(c - checkc) > 0.0001) {
    thePBErr._stream = stdout;
    thePBErr._type = PBErrTypeUnitTestFailed;
    sprintf(thePBErr._msg, "UnitTestIO failed");
    thePBErr._fatal = false;
    PBErrCatch(&thePBErr);
  }
  char checkd[10];
  PBErrScanf(&thePBErr, fd, "%s", checkd);
  if (strcmp(d, checkd) != 0) {
    thePBErr._stream = stdout;
    thePBErr._type = PBErrTypeUnitTestFailed;
    sprintf(thePBErr._msg, "UnitTestIO failed");
    thePBErr._fatal = false;
    PBErrCatch(&thePBErr);
  }
```

```
  PBErrCloseStream(&thePBErr, fd);
  fd = PBErrOpenStreamIn(&thePBErr, "./missingfile");
  printf("UnitTestIO OK\n");
}

void UnitTestCatch() {
  printf("UnitTestCatch\n");
  thePBErr._stream = stdout;
  thePBErr._type = PBErrTypeInvalidArg;
  sprintf(thePBErr._msg, "UnitTestCatch: invalid arg");
  thePBErr._fatal = false;
  PBErrCatch(&thePBErr);
  thePBErr._type = PBErrTypeNullPointer;
  sprintf(thePBErr._msg, "UnitTestCatch: null pointer");
  thePBErr._fatal = true;
  PBErrCatch(&thePBErr);
}

void fun() {
  if (isnan(0. / 0.)) Raise(TryCatchException_NaN);
}

void UnitTestTryCatch() {

  // --------------
  // Simple example, raise an exception in a TryCatch block and catch it.

  Try {

    if (isnan(0. / 0.)) Raise(TryCatchException_NaN);

  } Catch(TryCatchException_NaN) {

    printf("Catched exception NaN\n");

  } EndTry;

  // Output:
  //
  // Catched exception NaN
  //

  // --------------
  // Example of TryCatch block inside another TryCatch block and exception
  // forwarded from the inner block to the outer block after being ignored
  // by the inner block.

  Try {

    Try {

      if (isnan(0. / 0.)) Raise(TryCatchException_NaN);

    } EndTry;

  } Catch (TryCatchException_NaN) {

    printf("Catched exception NaN at sublevel\n");

  } EndTry;

  // Output:
```

```
  //
  // Catched exception NaN at sublevel
  //

  // --------------
  // Example of user defined exception and multiple catch segments.

  enum UserDefinedExceptions {

    myUserExceptionA = TryCatchException_LastID,
    myUserExceptionB,
    myUserExceptionC

  };

  Try {

    Raise(myUserExceptionA);

  } Catch (myUserExceptionA) {

    printf("Catched user defined exception A\n");

  } Catch (myUserExceptionB) {

    printf("Catched user defined exception B\n");

  } Catch (myUserExceptionC) {

    printf("Catched user defined exception C\n");

  } EndTry;

  // Output:
  //
  // Catched user defined exception A
  //

// The struct siginfo_t used to handle the SIGSEV is not defined in
// ANSI C, guard against this.
#ifndef __STRICT_ANSI__

  // --------------
  // Example of handling exception raided by SIGSEV.

  // Init the SIGSEV signal handling by TryCatch.
  TryCatchInitHandlerSigSegv();

  Try {

    int *p = NULL;
    *p = 1;

  } Catch (TryCatchException_Segv) {

    printf("Catched exception Segv\n");

  } EndTry;

  // Output:
  //
  // Catched exception Segv
```

```
  //
#endif

  // --------------
  // Example of exception raised in called function and catched in calling
  // function.

  Try {

    fun();

  } Catch (TryCatchException_NaN) {

    printf("Catched exception NaN raised in called function\n");

  } EndTry;

  // Output:
  //
  // Catched exception NaN raised in called function
  //

  // --------------
  // Example of exception raised in called function and uncatched in calling
  // function.

  Try {

    fun();

  } EndTry;

  // Output:
  //
  // Unhandled exception (2).
  //

  // --------------
  // Example of exception raised outside a TryCatch block.

  Raise(TryCatchException_NaN);

  // Output:
  //
  // Unhandled exception (2).
  //

  // --------------
  // Example of overflow of recursive inclusion of TryCatch blocks.

  Try {

    Try {

      Try {

        Try {

          fun();

        } EndTry;
```

```
      } EndTry;

    } EndTry;

  } EndTry;

  // Output:
  //
  // TryCatch blocks recursive incursion overflow, exiting. (You can try
  // to raise the value of TryCatchMaxExcLvl in trycatch.c, it was: 3)
  //

  printf("UnitTestTryCatch OK\n");
}

void UnitTestAll() {
  PBErrPrintln(&thePBErr, stdout);
  UnitTestCreateStatic();
  UnitTestReset();
  UnitTestMalloc();
  UnitTestIO();
  UnitTestTryCatch();
  UnitTestCatch();
}

int main(void) {
  UnitTestAll();
  return 0;
}
```

5 Unit tests output

```
main(PBErrCatch+0xd7)[0x55cc6f154cb7]
main(UnitTestAll+0xfb)[0x55cc6f154abb]
main(main+0xb)[0x55cc6f153e2b]
/lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xe7)[0x7fc1e930dbf7]
main(_start+0x2a)[0x55cc6f153e6a]
main(PBErrCatch+0xd7)[0x55cc6f154cb7]
main(main+0xb)[0x55cc6f153e2b]
/lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xe7)[0x7fc1e930dbf7]
main(_start+0x2a)[0x55cc6f153e6a]
PBErrFatal: true
UnitTestCreateStatic
PBErrFatal: true
UnitTestReset
Reset OK
UnitTestMalloc
Malloc OK
UnitTestIO OK
Catched exception NaN
Catched exception NaN at sublevel
Catched user defined exception
Catched exception Segv
Catched exception NaN in called function
UnitTestTryCatch OK
UnitTestCatch
---- PBErrCatch ----
PBErrType: invalid arguments
```

```
PBErrMsg: UnitTestCatch: invalid arg
PBErrFatal: false
Stack:
--------------------
---- PBErrCatch ----
PBErrType: null pointer
PBErrMsg: UnitTestCatch: null pointer
PBErrFatal: true
Stack:
Exiting
--------------------
```

testio.txt:

```
1
2
3.000000
string
```