

PBErr

P. Baillehache

August 2, 2020

Contents

1	Interface	1
2	Code	4
2.1	pberr.c	4
3	Makefile	13
4	Unit tests	13
5	Unit tests output	15

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>
```

```

#include <stdbool.h>
#include <execinfo.h>
#include <errno.h>
#include <string.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;

// ===== 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(PBERRSAFE_MALLOC)
    void* PBErrMalloc(PBErr* const that, const size_t size);
#else
    #define PBErrMalloc(That, Size) malloc(Size)
#endif

// Secured I/O
#if defined(PBERRALL) || defined(PBERRSAFE_IO)
    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, ...);

// ===== 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* PBImpAnalysisErr = &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;

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;
        }
    #endif
}

```

```

        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

```

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) {

```

```

        thePBEr._stream = stdout;
        thePBEr._type = PBErTypeUnitTestFailed;
        sprintf(thePBEr._msg, "UnitTestIO failed");
        thePBEr._fatal = false;
        PBErCatch(&thePBEr);
    }
    char checkd[10];
    PBErScanf(&thePBEr, fd, "%s", checkd);
    if (strcmp(d, checkd) != 0) {
        thePBEr._stream = stdout;
        thePBEr._type = PBErTypeUnitTestFailed;
        sprintf(thePBEr._msg, "UnitTestIO failed");
        thePBEr._fatal = false;
        PBErCatch(&thePBEr);
    }
    PBErCloseStream(&thePBEr, fd);
    fd = PBErOpenStreamIn(&thePBEr, "./missingfile");
    printf("UnitTestIO OK\n");
}

void UnitTestCatch() {
    printf("UnitTestCatch\n");
    thePBEr._stream = stdout;
    thePBEr._type = PBErTypeInvalidArg;
    sprintf(thePBEr._msg, "UnitTestCatch: invalid arg");
    thePBEr._fatal = false;
    PBErCatch(&thePBEr);
    thePBEr._type = PBErTypeNullPointer;
    sprintf(thePBEr._msg, "UnitTestCatch: null pointer");
    thePBEr._fatal = true;
    PBErCatch(&thePBEr);
}

void UnitTestAll() {
    PBErPrintln(&thePBEr, stdout);
    UnitTestCreateStatic();
    UnitTestReset();
    UnitTestMalloc();
    UnitTestIO();
    UnitTestCatch();
}

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

```

5 Unit tests output

```

main(PBErCatch+0xac) [0x804987c]
main(UnitTestAll+0xfd) [0x804970d]
main(main+0x16) [0x8048f46]
/lib/i386-linux-gnu/libc.so.6(__libc_start_main+0xf7) [0xb760f637]
main [0x8048f72]
main(PBErCatch+0xac) [0x804987c]
main(UnitTestAll+0x160) [0x8049770]
main(main+0x16) [0x8048f46]
/lib/i386-linux-gnu/libc.so.6(__libc_start_main+0xf7) [0xb760f637]

```

```

main[0x8048f72]
PBErrMsg:
PBErrFatal: true
UnitTestCreateStatic
PBErrMsg:
PBErrFatal: true
UnitTestReset
Reset OK
UnitTestMalloc
Malloc OK
UnitTestIO 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

```