

# PBErr

P. Baillehache

January 13, 2018

## Contents

<b>1</b>	<b>Interface</b>	<b>1</b>
<b>2</b>	<b>Code</b>	<b>3</b>
2.1	pberr.c . . . . .	3
<b>3</b>	<b>Makefile</b>	<b>4</b>
<b>4</b>	<b>Unit tests</b>	<b>5</b>
<b>5</b>	<b>Unit tests output</b>	<b>6</b>

## 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>

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

#define PBERR_MAXSTACKHEIGHT 10
#define PBERR_MSGLENGTHMAX 256

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

typedef enum PBErrType {
    PBErrTypeUnknown,
    PBErrTypeMallocFailed,
    PBErrTypeNullPointer,
    PBErrTypeInvalidArg,
    PBErrTypeUnitTestFailed,
    PBErrTypeOther,
    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;

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

// Static constructor
PBErr PBErrCreateStatic(void);

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

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

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

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

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

```

```
#endif
```

## 2 Code

### 2.1 pberr.c

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

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

#include "pberr.h"

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

PBErr thePBErr = {._msg[0] = '\0', ._type = PBErrTypeUnknown,
    ._stream = NULL, ._fatal = true};
PBErr* PBMathErr = &thePBErr;
PBErr* GSetErr = &thePBErr;
PBErr* ELORankErr = &thePBErr;

char *PBErrTypeLbl[PBErrTypeNb] = {
    "unknown",
    "malloc failed",
    "null pointer",
    "invalid arguments",
    "unit test failed",
    "other"
};

// ===== 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 *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 *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 (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(PBErr *that, FILE *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 != NULL)
        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 *that, size_t size) {
    void *ret = malloc(size);
    if (ret == NULL) {
        that->_type = PBErrTypeMallocFailed;
        sprintf(that->_msg, "malloc of %d bytes failed\n", size);
        that->_fatal = true;
        PBErrCatch(that);
    }
    return ret;
}
#endif

```

### 3 Makefile

```

# Build mode
# 0: development (max safety, no optimisation)
# 1: release (min safety, optimisation)
# 2: fast and furious (no safety, optimisation)
BUILDMODE=0

include ./Makefile.inc

BUILDOPTIONS=$(BUILDPARAM) $(INCPATH)

#rules
all : main

main: main.o pbeerr.o Makefile
$(COMPILER) main.o pbeerr.o $(LINKOPTIONS) -o main

```

```

main.o : main.c pberr.h Makefile
$(COMPILER) $(BUILDOPTIONS) -c main.c

pberr.o : pberr.c pberr.h Makefile
$(COMPILER) $(BUILDOPTIONS) -c pberr.c

clean :
rm -rf *.o main

valgrind :
valgrind -v --track-origins=yes --leak-check=full --gen-suppressions=yes --show-leak-kinds=all ./main

unitTest :
main > unitTest.txt; diff unitTest.txt unitTestRef.txt

```

## 4 Unit tests

```

#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.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 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 UnitTestAll() {
    PBErrPrintln(&thePBErr, stdout);
    UnitTestCreateStatic();
    UnitTestReset();
    UnitTestMalloc();
    UnitTestCatch();
}

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

```

## 5 Unit tests output

```

main(PBErrCatch+0x76) [0x8048f42]
main(UnitTestCatch+0x80) [0x8048d4e]
main(UnitTestAll+0x27) [0x8048ddc]
main(main+0x16) [0x8048df6]
/lib/i386-linux-gnu/libc.so.6(__libc_start_main+0xf7) [0xb7609637]
main [0x8048a51]
main(PBErrCatch+0x76) [0x8048f42]
main(UnitTestCatch+0xe3) [0x8048db1]
main(UnitTestAll+0x27) [0x8048ddc]
main(main+0x16) [0x8048df6]
/lib/i386-linux-gnu/libc.so.6(__libc_start_main+0xf7) [0xb7609637]
main [0x8048a51]
PBErrMsg:
PBErrFatal: true
UnitTestCreateStatic
PBErrMsg:
PBErrFatal: true
UnitTestReset
Reset OK
UnitTestMalloc
Malloc 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
-----
```