PBJson

P. Baillehache

July 25, 2019

Contents

Introduction

PBJson is a C library providing structures and functions to encode and decode data structures into JSON format.

An example is given below to show how the user can use PBJson to implement encoding and decoding functions of his/her data structures. Structures can include sub-structures recursively. Values can be atomic values (converted into string), array of atomic values, sub-structures, and array of sub-structures. The encoding can be done in a compact form (no indentation and no line return), or a readable form (indentation and line return). The decoding supports both compact and readable form. Keys and values

are delimited by double quote (") and values can include double quote by escaping them with an anti-slash (\). The library has the two folllowing limitations: key's label cannot starts with "[]", and the value must be less than 500 characters long.

It uses the `PBErr`, `GSet` and `GTree` libraries.

1 Definitions

Example of use:

```
// Declare two structures for example
struct structB {
  int _intVal;
};

struct structA {
  int _intVal;
  int _intArr[3];
  struct structB _structVal;
  struct structB _structArr[2];
};

// Function which return the JSON encoding of the structB 'that'
JSONNode* StructBEncodeAsJSON(struct structB* that) {

  // Create the JSON structure
  JSONNode* json = JSONCreate();

  // Declare a buffer to convert value into string
  char val[100];

  // Convert the value into string
  sprintf(val, "%d", that->_intVal);

  // Add a property to the JSON
  JSONAddProp(json, "_intVal", val);

  // Return the JSON
  return json;
}

// Function which return the JSON encoding of the structA 'that'
JSONNode* StructAEncodeAsJSON(struct structA* that) {

  // Create the JSON structure
  JSONNode* json = JSONCreate();

  // Declare a buffer to convert value into string
  char val[100];

  // Convert a int value into string
  sprintf(val, "%d", that->_intVal);
```

```
  // Add the property to the JSON
  JSONAddProp(json, "_intVal", val);

  // Declare an array of values converted to string
  JSONArrayVal setVal = JSONArrayValCreateStatic();
  // For each int value in the array
  for (int i = 0; i < 3; ++i) {
    // Convert the int value into string
    sprintf(val, "%d", that->_intArr[i]);
    // Add the string to the array
    JSONArrayValAdd(&setVal, val);
  }
  // Add the array of values to the JSON
  JSONAddProp(json, "_intArr", &setVal);

  // Add a key with an encoded structure as value
  JSONAddProp(json, "_structVal",
    StructBEncodeAsJSON(&(that->_structVal)));

  // Declare an array of structures converted to string
  JSONArrayStruct setStruct = JSONArrayStructCreateStatic();
  // Add the two encoded structures to the array
  JSONArrayStructAdd(&setStruct, StructBEncodeAsJSON(that->_structArr));
  JSONArrayStructAdd(&setStruct,
    StructBEncodeAsJSON(that->_structArr + 1));
  // Add a key with the array of structures
  JSONAddProp(json, "_structArr", &setStruct);

  // Free memory
  JSONArrayStructFlush(&setStruct);
  JSONArrayValFlush(&setVal);

  // Return the created JSON
  return json;
}

// Function which save the structA 'that' on the stream 'stream'
// If 'compact' equals true it saves in compact form, else it saves in
// readable form
void StructASave(struct structA* that, FILE* stream, bool compact) {

  // Get the JSON encoding of 'that'
  JSONNode* json = StructAEncodeAsJSON(that);

  // Save the JSON
  if (JSONSave(json, stream, compact) == false) {
    // ... manage the error
  }

  // Free memory
  JSONFree(&json);
}

// Function which decode from JSON encoding 'json' to the structB 'that'
bool StructBDecodeAsJSON(struct structB* that, JSONNode* json) {

  // Get the property _intVal from the JSON
  JSONNode* prop = JSONProperty(json, "_intVal");
  if (prop == NULL) {
    // ... manage the error
  }
```

```
  // Set the value of _intVal
  JSONNode* val = JSONValue(prop, 0);
  that->_intVal = atoi(JSONLabel(val));

  // Return the success code
  return true;
}

// Function which decode from JSON encoding 'json' to the structA 'that'
bool StructADecodeAsJSON(struct structA* that, JSONNode* json) {

  // Get the property _intVal from the JSON
  JSONNode* prop = JSONProperty(json, "_intVal");
  if (prop == NULL) {
    // ... manage the error
  }

  // Set the value of _intVal
  JSONNode* val = JSONValue(prop, 0);
  that->_intVal = atoi(JSONLabel(val));

  // Get the property _intArr from the JSON
  prop = JSONProperty(json, "_intArr");
  if (prop == NULL) {
    // ... manage the error
  }

  // Set the values of _intArr
  for (int i = 0; i < JSONGetNbValue(prop); ++i) {
    JSONNode* val = JSONValue(prop, i);
    that->_intArr[i] = atoi(JSONLabel(val));
  }

  // Get the property _structVal from the JSON
  prop = JSONProperty(json, "_structVal");
  if (prop == NULL) {
    // ... manage the error
  }

  // Decode the values of the sub struct
  if (StructBDecodeAsJSON(&(that->_structVal), prop) == false) {
    // ... manage the error
  }

  // Get the property _structArr from the JSON
  prop = JSONProperty(json, "_structArr");
  if (prop == NULL) {
    // ... manage the error
  }

  // Decode the values of _structArr
  for (int i = 0; i < JSONGetNbValue(prop); ++i) {
    JSONNode* val = JSONValue(prop, i);
    if (StructBDecodeAsJSON(that->_structArr + i, val) == false) {
      // ... manage the error
    }
  }

  // Return the success code
  return true;
}
```

```
// Function which load from the stream 'stream' containing the JSON
// encoding of the structA 'that'
void StructALoad(struct structA* that, FILE* stream) {

  // Declare a json to load the encoded data
  JSONNode* json = JSONCreate();

  // Load the whole encoded data
  if (JSONLoad(json, stream) == false) {
    // ... manage the error
  }

  // Decode the data from the JSON to the structA
  if (!StructADecodeAsJSON(that, json)) {
    // ... manage the error
  }

  // Free the memory used by the JSON
  JSONFree(&json);
}

// Create an instance of structA for example
struct structA myStruct;
myStruct._intVal = 1;
myStruct._intArr[0] = 2;
myStruct._intArr[1] = 3;
myStruct._intArr[2] = 4;
myStruct._structVal._intVal = 5;
myStruct._structArr[0]._intVal = 6;
myStruct._structArr[1]._intVal = 7;

// Save the structure in JSON encoding on the standard output stream
// in readable form
bool compact = false;
StructASave(&myStruct, stdout, compact);

// Load the structure in JSON encoding from the standard input stream
StructALoad(&myStruct, stdin);

// Result:

{
  "_intVal":"1",
  "_intArr":["2","3","4"],
  "_structVal":{
    "_intVal":"5"
  },
  "_structArr":[
    {
      "_intVal":"6"
    },
    {
      "_intVal":"7"
    }
  ]
}
```

2 Interface

```
// ============ PBJSON.H ================

#ifndef PBJSON_H
#define PBJSON_H

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

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdbool.h>
#include "pberr.h"
#include "gtree.h"

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

#define PBJSON_INDENT "  "
#define PBJSON_MAXLENGTHLBL 1024
#define PBJSON_CONTEXTSIZE 10

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

#define JSONNode GenTreeStr
#define JSONArrayVal GSetStr
#define JSONArrayStruct GSetGenTreeStr

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

// Free the memory used by the JSON node 'that' and its subnodes
// The memory used by the label of each node is freed too
void JSONFree(JSONNode** that);

// Set the label of the JSON node 'that' to a copy of 'lbl'
#if BUILDMODE != 0
inline
#endif
void JSONSetLabel(JSONNode* const that, const char* const lbl);

// Add a property to the node 'that'. The property's key is a copy of a
// 'key' and its value is a copy of 'val'
#if BUILDMODE != 0
inline
#endif
void _JSONAddPropStr(JSONNode* const that, const char* const key,
  char* const val);

// Add a property to the node 'that'. The property's key is a copy of a
// 'key' and its value is the JSON node 'val'
#if BUILDMODE != 0
inline
#endif
void _JSONAddPropObj(JSONNode* const that, const char* const key,
  JSONNode* const val);

// Add a property to the node 'that'. The property's key is a copy of a
// 'key' and its values are a copy of the values in the GSetStr 'set'
void _JSONAddPropArr(JSONNode* const that, const char* const key,
  const GSetStr* const set);
```

```
// Add a property to the node 'that'. The property's key is a copy of a
// 'key' and its values are the GenTreeStr in the GSetGenTreeStr 'set'
void _JSONAddPropArrObj(JSONNode* const that, const char* const key,
  const GSetGenTreeStr* const set);

// Save the JSON 'that' on the stream 'stream'
// If 'compact' equals true save in compact form, else save in easily
// readable form
// Return true if it could save, false else
bool JSONSave(const JSONNode* const that, FILE* const stream,
  const bool compact);

// Load the JSON 'that' from the stream 'stream'
// Return true if it could load, false else
bool JSONLoad(JSONNode* const that, FILE* const stream);

// Load the JSON 'that' from the string 'str' seen as a stream
// Return true if it could load, false else
bool JSONLoadFromStr(JSONNode* const that, const char* const str);

// Save the JSON 'that' in the string 'str' of length at least equal to
// 'strLen'
// If 'compact' equals true save in compact form, else save in easily
// readable form
// Return true if it could save, false else
bool JSONSaveToStr(const JSONNode* const that, char* const str,
  const size_t strLen, const bool compact);

// Return the JSONNode of the property with label 'lbl' of the
// JSON 'that'
// If the property doesn't exist return NULL
JSONNode* JSONProperty(const JSONNode* const that, const char* const lbl);

// Add a copy of the value 'val' to the array of value 'that'
#if BUILDMODE != 0
inline
#endif
void JSONArrayValAdd(JSONArrayVal* const that, const char* const val);

// Free memory used by the static array of values 'that'
#if BUILDMODE != 0
inline
#endif
void JSONArrayValFlush(JSONArrayVal* const that);

// Wrapping of GenTreeStr functions
#define JSONCreate() ((JSONNode*)GenTreeStrCreate())
#define JSONLabel(Node) GenTreeData(Node)
#define JSONAppendVal(Key, Val) GenTreeAppendSubtree(Key, Val)
#define JSONProperties(JSON) GenTreeSubtrees(JSON)
#define JSONValue(JSON, Index) GenTreeSubtree(JSON, Index)
#define JSONGetNbValue(JSON) GSetNbElem(GenTreeSubtrees(JSON))

// Wrapping of GSetStr functions
#define JSONArrayValCreateStatic() GSetStrCreateStatic()

// Wrapping of GSetGenTreeStr functions
#define JSONArrayStructCreateStatic() GSetGenTreeStrCreateStatic()
#define JSONArrayStructAdd(Array, Value) GSetAppend(Array, Value)
#define JSONArrayStructFlush(Array) GSetFlush(Array)
```

```
// Shortcut to get the label of the first value of the JSONNode 'node'
#define JSONLblVal(node) (JSONLabel(JSONValue(node, 0))

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

#define JSONAddProp(Node, Key, Val) _Generic(Val, \
  char*: _JSONAddPropStr, \
  const char*: _JSONAddPropStr, \
  JSONNode*: _JSONAddPropObj, \
  const JSONNode*: _JSONAddPropObj, \
  GSetStr*: _JSONAddPropArr, \
  const GSetStr*: _JSONAddPropArr, \
  GSetGenTreeStr*: _JSONAddPropArrObj, \
  const GSetGenTreeStr*: _JSONAddPropArrObj, \
  default: PBErrInvalidPolymorphism) (Node, Key, Val)

// ================ Inliner ====================

#if BUILDMODE != 0
#include "pbjson-inline.c"
#endif


#endif
```

3 Code

3.1 PBJson.c

```
// ============ PBJSON.C ================

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

#include "pbjson.h"
#if BUILDMODE == 0
#include "pbjson-inline.c"
#endif

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

// Save recursively the JSON tree 'that' into the stream 'stream'
// Return true if it could save, false else
bool JSONSaveRec(const JSONNode* const that, FILE* const stream,
  const bool compact, int depth);

// Return true if the JSON node 'that' is a value (ie its subtree is
// empty)
inline bool JSONIsValue(JSONNode* const that);

// Scan the 'stream' char by char until the next significant char
// ie anything else than a space or a new line or a tab and store the
// result in 'c'
// Return false if there has been an I/O error
inline bool JSONGetNextChar(FILE* stream, char* c);

// Load a struct in the JSON 'that' from the stream 'stream'
// Return true if it could load, false else
bool JSONLoadStruct(JSONNode* const that, FILE* stream);
```

```
// Load an array in the JSON 'that' from the stream 'stream'
// Return true if it could load, false else
bool JSONLoadArr(JSONNode* const that, FILE* stream, char* key);

// Load the string 'str' from the 'stream'
// Return false if there has been an I/O error
bool JSONLoadStr(FILE* stream, char* str);

// Load the array of values of property 'prop' in the JSON 'that'
// Return true if it could load, false else
bool JSONAddArr(JSONNode* const that, char* prop, FILE* stream);

// Load the array of structs of property 'prop' in the JSON 'that'
// Return true if it could load, false else
bool JSONAddArrStruct(JSONNode* const that, char* prop, FILE* stream);

// Get the characters around the current position in the 'stream'
void JSONGetContextStream(FILE* stream, char* buffer);

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

// Free the memory used by the JSON node 'that' and its subnodes
// The memory used by the label of each node is freed too
void JSONFree(JSONNode** that) {
  // Check arguments
  if (that == NULL || *that == NULL)
    // Nothing to do
    return;
  // Free all the char* in the tree
  if (JSONLabel(*that) != NULL)
    free(JSONLabel(*that));
  GenTreeIterDepth iter = GenTreeIterDepthCreateStatic((GenTreeStr*)(*that));
  if (!GenTreeIterIsLast(&iter)) {
    do {
      char* label = GenTreeIterGetData(&iter);
      free(label);
    } while (GenTreeIterStep(&iter));
  }
  GenTreeIterFreeStatic(&iter);
  // Free memory
  GenTreeFree(that);
}

// Add a property to the node 'that'. The property's key is a copy of a
// 'key' and its values are a copy of the values in the GSetStr 'set'
void _JSONAddPropArr(JSONNode* const that, const char* const key,
  const GSetStr* const set) {
#if BUILDMODE == 0
  if (that == NULL) {
    JSONErr->_type = PBErrTypeNullPointer;
    sprintf(JSONErr->_msg, "'that' is null");
    PBErrCatch(JSONErr);
  }
  if (key == NULL) {
    JSONErr->_type = PBErrTypeNullPointer;
    sprintf(JSONErr->_msg, "'key' is null");
    PBErrCatch(JSONErr);
  }
  if (set == NULL) {
    JSONErr->_type = PBErrTypeNullPointer;
    sprintf(JSONErr->_msg, "'set' is null");
```

```
    PBErrCatch(JSONErr);
  }
#endif
  // Create a new node for the key
  JSONNode* nodeKey = JSONCreate();
  // Set the key label
  JSONSetLabel(nodeKey, key);
  int nbElem = GSetNbElem(set);
  if (nbElem > 0) {
    // For each val in the set
    GSetIterForward iter = GSetIterForwardCreateStatic(set);
    do {
      // Get the value
      char* val = GSetIterGet(&iter);
      // Create a new node for the val
      JSONNode* nodeVal = JSONCreate();
      // Set the val label
      JSONSetLabel(nodeVal, val);
      // Attach the val to the key
      JSONAppendVal(nodeKey, nodeVal);
    } while (GSetIterStep(&iter));
  }
  // Add empty nodes to ensure it has at least 1 nodes and is viewed
  // as a property when saving
  while (nbElem < 1) {
    // Create a new empty node
    JSONNode* nodeVal = JSONCreate();
    // Attach the empty node to the key
    JSONAppendVal(nodeKey, nodeVal);
    ++nbElem;
  }
  // Attach the new property to the node 'that'
  JSONAppendVal(that, nodeKey);
}

// Add a property to the node 'that'. The property's key is a copy of a
// 'key' and its values are the GenTreeStr in the GSetGenTreeStr 'set'
void _JSONAddPropArrObj(JSONNode* const that, const char* const key,
  const GSetGenTreeStr* const set) {
  // Create a new node for the key
  JSONNode* nodeKey = JSONCreate();
  // Set the key label with '[]' as prefix
  char buffer[PBJSON_MAXLENGTHLBL + 3];
  buffer[0] = '[';buffer[1] = ']';
  sprintf(buffer + 2, "%s", key);
  JSONSetLabel(nodeKey, buffer);
  // GEt the number of value
  int nbElem = GSetNbElem(set);
  // If the array is not empty
  if (nbElem > 0) {
    // For each val in the set
    GSetIterForward iter = GSetIterForwardCreateStatic(set);
    do {
      // Get the value
      GenTreeStr* val = GSetIterGet(&iter);
      // Attach the val to the key
      JSONAppendVal(nodeKey, val);
    } while (GSetIterStep(&iter));
  }
  // Add empty nodes to ensure it has at least 1 nodes and is viewed
  // as a property when saving
  while (nbElem < 1) {
```

```
    // Create a new empty node
    JSONNode* nodeVal = JSONCreate();
    // Attach the empty node to the key
    JSONAppendVal(nodeKey, nodeVal);
    ++nbElem;
  }
  // Attach the new property to the node 'that'
  JSONAppendVal(that, nodeKey);
}

// Function to add indentation in beautiful mode
inline bool JSONIndent(FILE* stream, int depth) {
  for (int i = depth; i--;)
    if (!PBErrPrintf(JSONErr, stream, "%s", PBJSON_INDENT))
      return false;
  return true;
}

// Return true if the JSON node 'that' is a value (ie its subtree is
// empty)
inline bool JSONIsValue(JSONNode* const that) {
  return (GSetNbElem(GenTreeSubtrees(that)) == 0);
}

// Save the JSON 'that' in the string 'str' of length at least equal to
// 'strLen'
// If 'compact' equals true save in compact form, else save in easily
// readable form
// Return true if it could save, false else
bool JSONSaveToStr(const JSONNode* const that, char* const str,
  const size_t strLen, const bool compact) {
#if BUILDMODE == 0
  if (that == NULL) {
    JSONErr->_type = PBErrTypeNullPointer;
    sprintf(JSONErr->_msg, "'that' is null");
    PBErrCatch(JSONErr);
  }
  if (str == NULL) {
    JSONErr->_type = PBErrTypeNullPointer;
    sprintf(JSONErr->_msg, "'str' is null");
    PBErrCatch(JSONErr);
  }
#endif

  // Open the string as a stream
  FILE* stream = fmemopen((void*)str, strLen, "w");

  // Save the JSON as with a normal stream
  bool ret = JSONSave(that, stream, compact);
  fflush(stream);

  // Close the stream
  fclose(stream);

  // Return the success code
  return ret;
}

// Save the JSON 'that' on the stream 'stream'
// If 'compact' equals true save in compact form, else save in easily
// readable form
// Return true if it could save, false else
```

```
bool JSONSave(const JSONNode* const that, FILE* const stream,
  const bool compact) {
#if BUILDMODE == 0
  if (that == NULL) {
    JSONErr->_type = PBErrTypeNullPointer;
    sprintf(JSONErr->_msg, "'that' is null");
    PBErrCatch(JSONErr);
  }
  if (stream == NULL) {
    JSONErr->_type = PBErrTypeNullPointer;
    sprintf(JSONErr->_msg, "'stream' is null");
    PBErrCatch(JSONErr);
  }
#endif
  // Start the recursion at depth 0
  return JSONSaveRec(that, stream, compact, 0);
}

// Save recursively the JSON tree 'that' into the stream 'stream'
// Return true if it could save, false else
bool JSONSaveRec(const JSONNode* const that, FILE* const stream,
  const bool compact, int depth) {
  // Declare a flag to memorize if the current node is a key for an
  // array of object
  bool flagArrObj = false;
  // Declare a variable to memorize the opening and closing char
  char openChar[2] = "{";
  char closeChar[2] = "}";
  // Print the label of the property if it's not null
  if (JSONLabel(that) != NULL && strlen(JSONLabel(that)) > 0) {
    if (!compact && !JSONIndent(stream, depth))
      return false;
    char* lbl = JSONLabel(that);
    if (lbl[0] == '[' && lbl[1] == ']') {
      flagArrObj = true;
      lbl += 2;
      openChar[0] = '[';
      closeChar[0] = ']';
    }
    if (!PBErrPrintf(JSONErr, stream, "\"%s\":", lbl))
      return false;
  }
  // Loop on properties
  GSetIterForward iter =
    GSetIterForwardCreateStatic(JSONProperties(that));
  // Get the first property
  JSONNode* firstProp = GSetIterGet(&iter);
  // Declare a flag to escape opening and closing bracket in case of a
  // single array
  bool flagEscapeBracket = (depth == 0 &&
    GSetNbElem(JSONProperties(that)) == 1 &&
    (JSONLabel(firstProp) == NULL || strlen(JSONLabel(firstProp)) == 0));
  // Print the opening char if the first prop is not a value
  // It's enough to check on the first prop as the json is supposed
  // to be well formed, meaning if the first prop is not a value then
  // all the others too
  if (!JSONIsValue(firstProp) && !flagEscapeBracket) {
    if (!PBErrPrintf(JSONErr, stream, "%s", openChar))
      return false;
    if (!compact && !PBErrPrintf(JSONErr, stream, "%s", "\n"))
      return false;
    if (!compact && flagArrObj && !JSONIndent(stream, depth + 1))
```

```
      return false;
  }
  // Declare a flag to manage comma between values
  bool flagComma = false;
  // Loop on properties
  do {
    // Get the property
    JSONNode* prop = GSetIterGet(&iter);
    // If it's not a value (ie not a leaf)
    if (!JSONIsValue(prop)) {
      // Save the property's values
      if (!JSONSaveRec(prop, stream, compact, depth + 1))
        return false;
      if (!GSetIterIsLast(&iter)) {
        if (!PBErrPrintf(JSONErr, stream, "%s", ","))
          return false;
      }
      if (!compact && !PBErrPrintf(JSONErr, stream, "%s", "\n"))
        return false;
      if (!compact && flagArrObj && !GSetIterIsLast(&iter) &&
        !JSONIndent(stream, depth + 1))
        return false;
    // Else, it's a value
    } else {
      if (GSetNbElem(JSONProperties(that)) > 1 && GSetIterIsFirst(&iter))
        if (!PBErrPrintf(JSONErr, stream, "%s", "["))
          return false;
      if (flagComma) {
        if (!PBErrPrintf(JSONErr, stream, "%s", ","))
          return false;
      }
      if (JSONLabel(prop) != NULL) {
        if (!PBErrPrintf(JSONErr, stream, "\"%s\"",
          JSONLabel(prop)))
          return false;
      } else {
        if (!PBErrPrintf(JSONErr, stream, "%s", "\"\""))
          return false;
      }
      flagComma = true;
      if (GSetNbElem(JSONProperties(that)) > 1 && GSetIterIsLast(&iter))
        if (!PBErrPrintf(JSONErr, stream, "%s", "]"))
          return false;
    }
  } while (GSetIterStep(&iter));
  // Print the closing char if the first prop is not a value
  if (!JSONIsValue(firstProp) && !flagEscapeBracket) {
    if (!compact && !JSONIndent(stream, depth))
      return false;
    if (!PBErrPrintf(JSONErr, stream, "%s", closeChar))
      return false;
  }
  if (depth == 0 && !PBErrPrintf(JSONErr, stream, "%s", "\n"))
    return false;
  // Return the success code
  return true;
}

// Scan the 'stream' char by char until the next significant char
// ie anything else than a space or a new line or a tab or a comma
// and store the result in 'c'
// Return false if there has been an I/O error
```

```
inline bool JSONGetNextChar(FILE* stream, char* c) {
  // Loop until the next significant char
  do {
    // If we coudln't read the next character
    if (fscanf(stream, "%c", c) == EOF) {
      JSONErr->_type = PBErrTypeIOError;
      sprintf(JSONErr->_msg,
        "Premature end of file or fscanf error in JSONGetNextChar");
      return false;
    }
  } while (*c == ' ' || *c == '\n' || *c == '\t' || *c == ',');
  // Return the success code
  return true;
}

// Load the string 'str' from the 'stream'
// Return false if there has been an I/O error
bool JSONLoadStr(FILE* stream, char* str) {
  // Declare a variable ot memorize the position in the string
  int i = 0;
  // Declare a flag to manage escape character
  bool flagEsc = false;
  // Loop on character of the string
  do {
    // If the previous char was an escaped char
    if (flagEsc && i > 0 && str[i - 1] == '"')
      // Reset the flag
      flagEsc = false;
    // Read one character
    if (fscanf(stream, "%c", str + i) == EOF) {
      JSONErr->_type = PBErrTypeIOError;
      sprintf(JSONErr->_msg,
        "Premature end of file or fscanf error in JSONLoadStr");
      return false;
    }
    // If it's an escape char
    if (str[i] == '\\')
      // Set the flag
      flagEsc = true;
    // Increment the position in the string
    ++i;
  // Loop until the buffer is full or we reached the final double quote
  } while ((flagEsc || str[i - 1] != '"') && i < PBJSON_MAXLENGTHLBL);
  // Add the null character at the end of the string
  str[i - 1] = '\0';
  // Return the success code
  return true;
}

// Load the array of values of property 'prop' in the JSON 'that'
// Return true if it could load, false else
bool JSONAddArr(JSONNode* const that, char* prop, FILE* stream) {
  // Declare the array of values
  JSONArrayVal set = JSONArrayValCreateStatic();
  // Declare a buffer for the value
  char bufferValue[PBJSON_MAXLENGTHLBL + 1] = {'\0'};
  // Declare a char to memorize the next significant char
  char c = '\0';
  // Loop on values
  do {
    // Load the value
    if (!JSONLoadStr(stream, bufferValue))
```

```
      return false;
    // Add the string to the array
    JSONArrayValAdd(&set, bufferValue);
    // Move to the next significant char
    if (!JSONGetNextChar(stream, &c))
      return false;
    // Check the next significant character is '"' or ']'
    if (c != '"' && c != ']') {
      JSONErr->_type = PBErrTypeInvalidData;
      char ctx[2 * PBJSON_CONTEXTSIZE + 1];
      JSONGetContextStream(stream, ctx);
      sprintf(JSONErr->_msg,
        "JSONAddArr: Expected '\"' or ']' but found '%c' near ...%s...",
        c, ctx);
      return false;
    }
  } while (c != ']');
  // Add the property to the JSON
  JSONAddProp(that, prop, &set);
  // Flush the array
  JSONArrayValFlush(&set);
  // Return the success code
  return true;
}

// Load the array of structs of property 'prop' in the JSON 'that'
// Return true if it could load, false else
bool JSONAddArrStruct(JSONNode* const that, char* prop, FILE* stream) {
  // Declare the array of values
  JSONArrayStruct set = JSONArrayStructCreateStatic();
  // Declare a char to memorize the next significant char
  char c = '\0';
  // Loop on values
  do {
    // Allocate memory for the next object
    JSONNode* obj = JSONCreate();
    // Rewind one char as JSONLoad expect to read '{'
    if (fseek(stream, -1, SEEK_CUR) != 0) {
      JSONErr->_type = PBErrTypeIOError;
      sprintf(JSONErr->_msg, "fseek error in JSONAddArrStruct");
      return false;
    }
    // Load the value
    if (!JSONLoad(obj, stream))
      return false;
    // Add the string to the array
    JSONArrayStructAdd(&set, obj);
    // Move the next significant char
    if (!JSONGetNextChar(stream, &c))
      return false;
    // check the next significant character is '{' or ']'
    if (c != '{' && c != ']') {
      JSONErr->_type = PBErrTypeInvalidData;
      char ctx[2 * PBJSON_CONTEXTSIZE + 1];
      JSONGetContextStream(stream, ctx);
      sprintf(JSONErr->_msg,
        "JSONAddStruct: Expected '{' or ']' but found '%c' near ...%s...",
        c, ctx);
      return false;
    }
  } while (c != ']');
  // Add the property to the JSON
```

```
  JSONAddProp(that, prop, &set);
  // Flush the array
  JSONArrayStructFlush(&set);
  // Return the success code
  return true;
}

// Load a key/value in the JSON 'that' from the stream 'stream'
// Return true if it could load, false else
bool JSONLoadProp(JSONNode* const that, FILE* stream) {
  // Declare a buffer to read the key
  char bufferKey[PBJSON_MAXLENGTHLBL + 1] = {'\0'};
  // Read the property's key
  if (!JSONLoadStr(stream, bufferKey))
    return false;
  // Read the next significant character which must be a ':'
  char c;
  if (!JSONGetNextChar(stream, &c))
    return false;
  if (c != ':') {
    JSONErr->_type = PBErrTypeInvalidData;
    char ctx[2 * PBJSON_CONTEXTSIZE + 1];
    JSONGetContextStream(stream, ctx);
    sprintf(JSONErr->_msg,
      "JSONLoadProp: Expected ':' but found '%c' near ...%s...",
      c, ctx);
    return false;
  }
  // Read the next significant character
  if (!JSONGetNextChar(stream, &c))
    return false;
  // If the next character is a double quote
  if (c == '"') {
    // Read the property's value
    char bufferVal[PBJSON_MAXLENGTHLBL + 1] = {'\0'};
    if (!JSONLoadStr(stream, bufferVal))
      return false;
    // Add the property to the JSON
    JSONAddProp(that, bufferKey, bufferVal);
  // Else, if the next character is a square bracket
  } else if (c == '[') {
    JSONLoadArr(that, stream, bufferKey);
  // Else, if the next character is an accolade
  } else if (c == '{') {
    // This property is an object
    // Create a new node for the object
    JSONNode* prop = JSONCreate();
    // Set the property name
    JSONSetLabel(prop, bufferKey);
    // Add the new node to the JSON
    JSONAppendVal(that, prop);
    // Load the object
    JSONLoadStruct(prop, stream);
  // Else, it's not a valid file
  } else {
    // Return the failure code
    JSONErr->_type = PBErrTypeInvalidData;
    char ctx[2 * PBJSON_CONTEXTSIZE + 1];
    JSONGetContextStream(stream, ctx);
    sprintf(JSONErr->_msg,
      "JSONLoadProp: Expected '\"','{' or '[' but found '%c' near ...%s...",
      c, ctx);
```

```
    return false;
  }
  // Return the success code
  return true;
}

// Get the characters around the current position in the 'stream'
void JSONGetContextStream(FILE* stream, char* buffer) {
  int pos = fseek(stream, -PBJSON_CONTEXTSIZE, SEEK_CUR);
  (void)pos;
  int nb = fread(buffer, sizeof(char), 2 * PBJSON_CONTEXTSIZE, stream);
  (void)nb;
  buffer[2 * PBJSON_CONTEXTSIZE] = '\0';
}

// Load a struct in the JSON 'that' from the stream 'stream'
// Return true if it could load, false else
bool JSONLoadStruct(JSONNode* const that, FILE* stream) {
  char c = '\0';
  // Loop until the end of the structure
  while (c != '}') {
    // Read the next significant character
    if (!JSONGetNextChar(stream, &c))
      return false;
    // If it's not the end of the struct
    if (c != '}') {
      // Load the pair key/value
      if (!JSONLoadProp(that, stream))
        return false;
    }
  }
  // Return the success code
  return true;
}

// Load an array in the JSON 'that' from the stream 'stream'
// Return true if it could load, false else
bool JSONLoadArr(JSONNode* const that, FILE* stream, char* key) {
  // Declare a variable ot memorize the next significant char
  char c;
  // Read the next significant character
  if (!JSONGetNextChar(stream, &c))
    return false;
  // If the next character is a double quote
  if (c == '"') {
    // Load the array of value
    if (!JSONAddArr(that, key, stream))
      return false;
  // Else, if the next character is a closing square bracket
  } else if (c == ']') {
    // It's an empty array
    // Declare the empty array
    JSONArrayVal set = JSONArrayValCreateStatic();
    // Add the property to the JSON
    JSONAddProp(that, key, &set);
  // Else, if the next character is a bracket
  } else if (c == '{') {
    // This property is an array of structs
    // Load the array of structs
    if (!JSONAddArrStruct(that, key, stream))
      return false;
  // Else, it's not a valid file
```

```
  } else {
    // Return the failure code
    JSONErr->_type = PBErrTypeInvalidData;
    char ctx[2 * PBJSON_CONTEXTSIZE + 1];
    JSONGetContextStream(stream, ctx);
    sprintf(JSONErr->_msg,
      "JSONLoadArr: Expected '\"' or '{' but found '%c' near ...%s...",
      c, ctx);
    return false;
  }
  // Return the success code
  return true;
}

// Load the JSON 'that' from the stream 'stream'
// Return true if it could load, false else
bool JSONLoad(JSONNode* const that, FILE* const stream) {
#if BUILDMODE == 0
  if (that == NULL) {
    JSONErr->_type = PBErrTypeNullPointer;
    sprintf(JSONErr->_msg, "'that' is null");
    PBErrCatch(JSONErr);
  }
  if (stream == NULL) {
    JSONErr->_type = PBErrTypeNullPointer;
    sprintf(JSONErr->_msg, "'stream' is null");
    PBErrCatch(JSONErr);
  }
#endif
  char c;
  // Read the first significant character
  if (!JSONGetNextChar(stream, &c))
    return false;
  // If the file starts with a '{'
  if (c == '{') {
    // The file contains a struct definion
    // Load the struct
    return JSONLoadStruct(that, stream);
  // Else if the file starts with a '['
  } else if (c == '[') {
    // The file contains an array
    // Load the array
    return JSONLoadArr(that, stream, "");
  // Else, the file doesn't start with '{' or '['
  } else {
    // It's not a valid file, stop here
    JSONErr->_type = PBErrTypeInvalidData;
    char ctx[2 * PBJSON_CONTEXTSIZE + 1];
    JSONGetContextStream(stream, ctx);
    sprintf(JSONErr->_msg,
      "JSONLoad: Expected '{' or '[' but found '%c' near ...%s...",
      c, ctx);
    return false;
  }
  // Return the success code
  return true;
}

// Load the JSON 'that' from the string 'str' seen as a stream
// Return true if it could load, false else
bool JSONLoadFromStr(JSONNode* const that, const char* const str) {
#if BUILDMODE == 0
```

```
  if (that == NULL) {
    JSONErr->_type = PBErrTypeNullPointer;
    sprintf(JSONErr->_msg, "'that' is null");
    PBErrCatch(JSONErr);
  }
  if (str == NULL) {
    JSONErr->_type = PBErrTypeNullPointer;
    sprintf(JSONErr->_msg, "'str' is null");
    PBErrCatch(JSONErr);
  }
#endif
  // Open the string as a stream
  FILE* stream = fmemopen((void*)str, strlen(str), "r");

  // Load the JSON as with a normal stream
  bool ret = JSONLoad(that, stream);

  // Close the stream
  fclose(stream);

  // Return the success code
  return ret;
}

// Return the JSONNode of the property with label 'lbl' of the
// JSON 'that'
// If the property doesn't exist return NULL
JSONNode* JSONProperty(const JSONNode* const that,
  const char* const lbl) {
#if BUILDMODE == 0
  if (that == NULL) {
    JSONErr->_type = PBErrTypeNullPointer;
    sprintf(JSONErr->_msg, "'that' is null");
    PBErrCatch(JSONErr);
  }
  if (lbl == NULL) {
    JSONErr->_type = PBErrTypeNullPointer;
    sprintf(JSONErr->_msg, "'lbl' is null");
    PBErrCatch(JSONErr);
  }
#endif
  // If the JSONNode has properties
  if (JSONGetNbValue(that) > 0) {
    // Declare an iterator on properties of the JSONNode
    GSetIterForward iter =
      GSetIterForwardCreateStatic(JSONProperties(that));
    // Loop on properties
    do {
      // Get the property
      JSONNode* prop = GSetIterGet(&iter);
      // Skip the eventual '[]'
      char* propLbl = JSONLabel(prop);
      if (propLbl[0] == '[' && propLbl[1] == ']')
        propLbl += 2;
      // If the label of the property is the same as the searched
      // property
      if (strcmp(propLbl, lbl) == 0) {
        // Return the property
        return prop;
      }
    } while (GSetIterStep(&iter));
  }
```

```
  // If we reach here it means the searched property doesn't exist
  return NULL;
}
```

3.2 PBJson-inline.c

```
// ============ PBJSON-INLINE.C ================

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

// Set the label of the JSON node 'that' to a copy of 'lbl'
#if BUILDMODE != 0
inline
#endif
void JSONSetLabel(JSONNode* const that, const char* const lbl) {
#if BUILDMODE == 0
  if (that == NULL) {
    JSONErr->_type = PBErrTypeNullPointer;
    sprintf(JSONErr->_msg, "'that' is null");
    PBErrCatch(JSONErr);
  }
  if (lbl == NULL) {
    JSONErr->_type = PBErrTypeNullPointer;
    sprintf(JSONErr->_msg, "'lbl' is null");
    PBErrCatch(JSONErr);
  }
#endif
  // If the node already as a label
  if (JSONLabel(that) != NULL)
    // Free the label
    free(JSONLabel(that));
  // Allocate memory for the new label
  char* str = PBErrMalloc(JSONErr, sizeof(char) * (1 + strlen(lbl)));
  // Set the label copy
  strcpy(str, lbl);
  GenTreeSetData(that, str);
}

// Add a property to the node 'that'. The property's key is a copy of a
// 'key' and its value is a copy of 'val'
#if BUILDMODE != 0
inline
#endif
void _JSONAddPropStr(JSONNode* const that, const char* const key,
  char* const val) {
#if BUILDMODE == 0
  if (that == NULL) {
    JSONErr->_type = PBErrTypeNullPointer;
    sprintf(JSONErr->_msg, "'that' is null");
    PBErrCatch(JSONErr);
  }
  if (key == NULL) {
    JSONErr->_type = PBErrTypeNullPointer;
    sprintf(JSONErr->_msg, "'key' is null");
    PBErrCatch(JSONErr);
  }
  if (val == NULL) {
    JSONErr->_type = PBErrTypeNullPointer;
    sprintf(JSONErr->_msg, "'val' is null");
```

```
    PBErrCatch(JSONErr);
  }
#endif
  // Create a new node for the key
  JSONNode* nodeKey = JSONCreate();
  // Create a new node for the val
  JSONNode* nodeVal = JSONCreate();
  // Set the key and val label
  JSONSetLabel(nodeKey, key);
  JSONSetLabel(nodeVal, val);
  // Attach the val to the key
  JSONAppendVal(nodeKey, nodeVal);
  // Attach the new property to the node 'that'
  JSONAppendVal(that, nodeKey);
}

// Add a property to the node 'that'. The property's key is a copy of a
// 'key' and its value is the JSON node 'val'
#if BUILDMODE != 0
inline
#endif
void _JSONAddPropObj(JSONNode* const that, const char* const key,
  JSONNode* const val) {
#if BUILDMODE == 0
  if (that == NULL) {
    JSONErr->_type = PBErrTypeNullPointer;
    sprintf(JSONErr->_msg, "'that' is null");
    PBErrCatch(JSONErr);
  }
  if (key == NULL) {
    JSONErr->_type = PBErrTypeNullPointer;
    sprintf(JSONErr->_msg, "'key' is null");
    PBErrCatch(JSONErr);
  }
  if (val == NULL) {
    JSONErr->_type = PBErrTypeNullPointer;
    sprintf(JSONErr->_msg, "'val' is null");
    PBErrCatch(JSONErr);
  }
#endif
  // Set the key label for the node value
  JSONSetLabel(val, key);
  // Attach the value to the node 'that'
  JSONAppendVal(that, val);
}

// Add a copy of the value 'val' to the array of value 'that'
#if BUILDMODE != 0
inline
#endif
void JSONArrayValAdd(JSONArrayVal* const that, const char* const val) {
#if BUILDMODE == 0
  if (that == NULL) {
    JSONErr->_type = PBErrTypeNullPointer;
    sprintf(JSONErr->_msg, "'that' is null");
    PBErrCatch(JSONErr);
  }
  if (val == NULL) {
    JSONErr->_type = PBErrTypeNullPointer;
    sprintf(JSONErr->_msg, "'val' is null");
    PBErrCatch(JSONErr);
  }
```

```
#endif
  // Create a copy of the value
  char* lbl = PBErrMalloc(JSONErr, sizeof(char) * (1 + strlen(val)));
  strcpy(lbl, val);
  // Add the copy to the set
  GSetAppend(that, lbl);
}

// Free memory used by the static array of values 'that'
#if BUILDMODE != 0
inline
#endif
void JSONArrayValFlush(JSONArrayVal* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    JSONErr->_type = PBErrTypeNullPointer;
    sprintf(JSONErr->_msg, "'that' is null");
    PBErrCatch(JSONErr);
  }
#endif
  // Free the memory used by the values
  while (GSetNbElem(that) > 0) {
    char* val = GSetPop(that);
    free(val);
  }
}
```

4 Makefile

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

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=pbjson
$($(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)/$
```

5 Unit tests

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/time.h>
#include "pberr.h"
#include "pbjson.h"

#define RANDOMSEED 0

void UnitTestJSONCreateFree() {
  JSONNode* json = JSONCreate();
  if (json == NULL) {
    JSONErr->_type = PBErrTypeUnitTestFailed;
    sprintf(JSONErr->_msg, "JSONCreate failed");
    PBErrCatch(JSONErr);
  }
  JSONFree(&json);
  if (json != NULL) {
    JSONErr->_type = PBErrTypeUnitTestFailed;
    sprintf(JSONErr->_msg, "JSONFree failed");
    PBErrCatch(JSONErr);
  }
  printf("UnitTestJSONCreateFree OK\n");
}

void UnitTestJSONSetGet() {
  JSONNode* json = JSONCreate();
  char* lbl = "testlabel";
  JSONSetLabel(json, lbl);
  if (strcmp(lbl, JSONLabel(json)) != 0) {
    JSONErr->_type = PBErrTypeUnitTestFailed;
    sprintf(JSONErr->_msg, "JSONSetLabel failed");
    PBErrCatch(JSONErr);
  }
  char* key = "key";
  char* val = "val";
  JSONAddProp(json, key, val);
  if (strcmp(key, JSONLabel(GenTreeSubtree(json, 0))) != 0 ||
    strcmp(val,
      JSONLabel(GenTreeSubtree(GenTreeSubtree(json, 0), 0))) != 0) {
    JSONErr->_type = PBErrTypeUnitTestFailed;
    sprintf(JSONErr->_msg, "JSONAddProp failed");
    PBErrCatch(JSONErr);
  }
  JSONNode* prop = JSONCreate();
  JSONAddProp(prop, key, val);
  char* propkey = "propkey";
  JSONAddProp(json, propkey, prop);
  if (strcmp(propkey, JSONLabel(prop)) != 0 ||
    GenTreeSubtree(json, 1) != prop) {
    JSONErr->_type = PBErrTypeUnitTestFailed;
    sprintf(JSONErr->_msg, "JSONAddProp failed");
    PBErrCatch(JSONErr);
  }
  JSONFree(&json);
  printf("UnitTestJSONSetGet OK\n");
```

```
}

struct structB {
  int _intVal;
  float _floatVal;
};

struct structA {
  int _intVal;
  int _intArr[3];
  struct structB _structVal;
  struct structB _structArr[2];
};

JSONNode* StructBEncodeAsJSON(struct structB* that) {
  // Create the JSON structure
  JSONNode* json = JSONCreate();
  // Declare a buffer to convert value into string
  char val[100];
  // Convert the value into string
  sprintf(val, "%d", that->_intVal);
  // Add a key/value to the JSON
  JSONAddProp(json, "_intVal", val);
  // Convert the value into string
  sprintf(val, "%f", that->_floatVal);
  // Add a key/value to the JSON
  JSONAddProp(json, "_floatVal", val);
  // Return the JSON
  return json;
}

JSONNode* StructAEncodeAsJSON(struct structA* that) {
  // Create the JSON structure
  JSONNode* json = JSONCreate();
  // Declare a buffer to convert value into string
  char val[100];

  // Add a property with empty value to the JSON
  val[0] = '\0';
  JSONAddProp(json, "_emptyVal", val);

  // Convert a int value into string
  sprintf(val, "%d", that->_intVal);
  // Add the property to the JSON
  JSONAddProp(json, "_intVal", val);

  // Add a property with a value containing a double quote to the JSON
  sprintf(val, "\\\"double quoted\\\"");
  JSONAddProp(json, "_escapeVal", val);

  // Declare an array of values converted to string
  JSONArrayVal setVal = JSONArrayValCreateStatic();
  // Create buffer for the conversion of int values into strinng
  char valInt[100];
  // For each int value in the array
  for (int i = 0; i < 3; ++i) {
    // Convert the int value into string
    sprintf(valInt, "%d", that->_intArr[i]);
    // Add the string to the array
    JSONArrayValAdd(&setVal, valInt);
  }
  // Add a property with the array of values to the JSON
```

```
  JSONAddProp(json, "_intArr", &setVal);

  // Empty the array
  JSONArrayValFlush(&setVal);
  // Add a property with an empty array of value
  JSONAddProp(json, "_emptyArr", &setVal);

  // Put back one value in the array
  JSONArrayValAdd(&setVal, valInt);
  // Add a property with an array of only one value
  JSONAddProp(json, "_oneIntArr", &setVal);

  // Add a property with an encoded structure as value
  JSONAddProp(json, "_structVal",
    StructBEncodeAsJSON(&(that->_structVal)));

  // Declare an array of structures converted to string
  JSONArrayStruct setStruct = JSONArrayStructCreateStatic();
  // Add the two encoded structures to the array
  JSONArrayStructAdd(&setStruct, StructBEncodeAsJSON(that->_structArr));
  JSONArrayStructAdd(&setStruct,
    StructBEncodeAsJSON(that->_structArr + 1));
  // Add a property with the array of structures
  JSONAddProp(json, "_structArr", &setStruct);

  // Free memory
  JSONArrayStructFlush(&setStruct);
  JSONArrayValFlush(&setVal);

  // Return the created JSON
  return json;
}

void StructASave(struct structA* that, FILE* stream, bool compact) {
  // Get the JSON encoding of 'that'
  JSONNode* json = StructAEncodeAsJSON(that);

  // Save the JSON
  if (JSONSave(json, stream, compact) == false) {
    JSONErr->_type = PBErrTypeUnitTestFailed;
    sprintf(JSONErr->_msg, "JSONSave failed");
    PBErrCatch(JSONErr);
  }

  // Free memory
  JSONFree(&json);
}

bool StructBDecodeAsJSON(struct structB* that, JSONNode* json) {
  // Get the property _intVal from the JSON
  JSONNode* prop = JSONProperty(json, "_intVal");
  if (prop == NULL) {
    JSONErr->_type = PBErrTypeUnitTestFailed;
    sprintf(JSONErr->_msg, "StructBDecodeAsJSON failed");
    PBErrCatch(JSONErr);
  }
  // Set the value of _intVal
  JSONNode* val = JSONValue(prop, 0);
  that->_intVal = atoi(JSONLabel(val));
  // Get the property _floatVal from the JSON
  prop = JSONProperty(json, "_floatVal");
  if (prop == NULL) {
```

```
    JSONErr->_type = PBErrTypeUnitTestFailed;
    sprintf(JSONErr->_msg, "StructBDecodeAsJSON failed");
    PBErrCatch(JSONErr);
  }
  // Set the value of _floatVal
  val = JSONValue(prop, 0);
  that->_floatVal = atof(JSONLabel(val));
  // Return the success code
  return true;
}

bool StructADecodeAsJSON(struct structA* that, JSONNode* json) {
  // Get the property _intVal from the JSON
  JSONNode* prop = JSONProperty(json, "_intVal");
  if (prop == NULL) {
    JSONErr->_type = PBErrTypeUnitTestFailed;
    sprintf(JSONErr->_msg, "StructADecodeAsJSON failed");
    PBErrCatch(JSONErr);
  }
  // Set the value of _intVal
  JSONNode* val = JSONValue(prop, 0);
  that->_intVal = atoi(JSONLabel(val));

  // Get the property _intArr from the JSON
  prop = JSONProperty(json, "_intArr");
  if (prop == NULL) {
    JSONErr->_type = PBErrTypeUnitTestFailed;
    sprintf(JSONErr->_msg, "StructADecodeAsJSON failed");
    PBErrCatch(JSONErr);
  }
  // Set the values of _intArr
  for (int i = 0; i < JSONGetNbValue(prop); ++i) {
    JSONNode* val = JSONValue(prop, i);
    that->_intArr[i] = atoi(JSONLabel(val));
  }

  // Get the property _structVal from the JSON
  prop = JSONProperty(json, "_structVal");
  if (prop == NULL) {
    JSONErr->_type = PBErrTypeUnitTestFailed;
    sprintf(JSONErr->_msg, "StructADecodeAsJSON failed");
    PBErrCatch(JSONErr);
  }
  // Decode the values of the sub struct
  if (StructBDecodeAsJSON(&(that->_structVal), prop) == false) {
    JSONErr->_type = PBErrTypeUnitTestFailed;
    sprintf(JSONErr->_msg, "StructBDecodeAsJSON failed");
    PBErrCatch(JSONErr);
  }

  // Get the property _structArr from the JSON
  prop = JSONProperty(json, "_structArr");
  if (prop == NULL) {
    JSONErr->_type = PBErrTypeUnitTestFailed;
    sprintf(JSONErr->_msg, "StructADecodeAsJSON failed");
    PBErrCatch(JSONErr);
  }
  // Decode the values of _structArr
  for (int i = 0; i < JSONGetNbValue(prop); ++i) {
    JSONNode* val = JSONValue(prop, i);
    if (StructBDecodeAsJSON(that->_structArr + i, val) == false) {
      JSONErr->_type = PBErrTypeUnitTestFailed;
```

```
      sprintf(JSONErr->_msg, "StructBDecodeAsJSON failed");
      PBErrCatch(JSONErr);
    }
  }

  // Return the success code
  return true;
}

void StructALoad(struct structA* that, FILE* stream) {
  // Declare a json to load the encoded data
  JSONNode* json = JSONCreate();
  // Load the whole encoded data
  if (JSONLoad(json, stream) == false) {
    JSONErr->_type = PBErrTypeUnitTestFailed;
    sprintf(JSONErr->_msg, "JSONLoad failed");
    PBErrCatch(JSONErr);
  }
  // Decode the data from the JSON to the structA
  if (!StructADecodeAsJSON(that, json)) {
    JSONErr->_type = PBErrTypeUnitTestFailed;
    sprintf(JSONErr->_msg, "StructADecodeAsJSON failed");
    PBErrCatch(JSONErr);
  }
  // Free the memory used by the JSON
  JSONFree(&json);
}

void UnitTestJSONLoadSave() {
  struct structA myStruct;
  myStruct._intVal = 1;
  myStruct._intArr[0] = 2;
  myStruct._intArr[1] = 3;
  myStruct._intArr[2] = 4;
  myStruct._structVal._intVal = 5;
  myStruct._structVal._floatVal = 6.0;
  myStruct._structArr[0]._intVal = 7;
  myStruct._structArr[0]._floatVal = 8.0;
  myStruct._structArr[1]._intVal = 9;
  myStruct._structArr[1]._floatVal = 10.0;
  bool compact = false;
  printf("myStruct:\n");
  StructASave(&myStruct, stdout, compact);
  FILE* fd = fopen("./testJsonReadable.txt", "w");
  StructASave(&myStruct, fd, compact);
  fclose(fd);
  compact = true;
  fd = fopen("./testJsonCompact.txt", "w");
  StructASave(&myStruct, fd, compact);
  fclose(fd);

  struct structA myStructLoad;
  fd = fopen("./testJsonReadable.txt", "r");
  StructALoad(&myStructLoad, fd);
  fclose(fd);
  if (memcmp(&myStructLoad, &myStruct, sizeof(struct structA)) != 0) {
    JSONErr->_type = PBErrTypeUnitTestFailed;
    sprintf(JSONErr->_msg, "JSONLoad failed");
    PBErrCatch(JSONErr);
  }

  char* array[3] = {"8","9","10"};
```

```
  JSONNode* json = JSONCreate();
  JSONArrayVal set = JSONArrayValCreateStatic();
  for (int i = 0; i < 3; ++i)
    JSONArrayValAdd(&set, array[i]);
  JSONAddProp(json, "", &set);
  JSONArrayValFlush(&set);
  printf("array:\n");
  if (!JSONSave(json, stdout, true)) {
    JSONErr->_type = PBErrTypeUnitTestFailed;
    sprintf(JSONErr->_msg, "JSONSave failed");
    PBErrCatch(JSONErr);
  }
  fd = fopen("./testJsonArray.txt", "w");
  if (!JSONSave(json, fd, false)) {
    JSONErr->_type = PBErrTypeUnitTestFailed;
    sprintf(JSONErr->_msg, "JSONSave failed");
    PBErrCatch(JSONErr);
  }
  fclose(fd);
  fd = fopen("./testJsonArray.txt", "r");
  JSONNode* jsonLoaded = JSONCreate();
  if (JSONLoad(jsonLoaded, fd) == false) {
    JSONErr->_type = PBErrTypeUnitTestFailed;
    sprintf(JSONErr->_msg, "JSONLoad failed");
    PBErrCatch(JSONErr);
  }
  if (strcmp(JSONLabel(JSONValue(JSONValue(jsonLoaded, 0), 0)),
    array[0]) != 0 ||
    strcmp(JSONLabel(JSONValue(JSONValue(jsonLoaded, 0), 1)),
    array[1]) != 0 ||
    strcmp(JSONLabel(JSONValue(JSONValue(jsonLoaded, 0), 2)),
    array[2]) != 0) {
    JSONErr->_type = PBErrTypeUnitTestFailed;
    sprintf(JSONErr->_msg, "JSONLoad failed");
    PBErrCatch(JSONErr);
  }
  fclose(fd);
  JSONFree(&json);
  JSONFree(&jsonLoaded);
  JSONNode* jsonStr = JSONCreate();
  char* str = "{\"v\":\"1\"}\n";
  if (JSONLoadFromStr(jsonStr, str) == false) {
    JSONErr->_type = PBErrTypeUnitTestFailed;
    sprintf(JSONErr->_msg, "JSONLoadFromStr failed");
    PBErrCatch(JSONErr);
  }
  char strSave[50] = {0};
  if (JSONSaveToStr(jsonStr, strSave, 50, true) == false ||
    strcmp(str, strSave) != 0) {
    JSONErr->_type = PBErrTypeUnitTestFailed;
    sprintf(JSONErr->_msg, "JSONSaveToStr failed");
    PBErrCatch(JSONErr);
  }

  JSONFree(&jsonStr);
  printf("UnitTestJSONLoadSave OK\n");
}

void UnitTestJSON() {
  UnitTestJSONCreateFree();
  UnitTestJSONSetGet();
  UnitTestJSONLoadSave();
```

```
  printf("UnitTestJSON OK\n");
}

void UnitTestAll() {
  UnitTestJSON();
  printf("UnitTestAll OK\n");
}

int main() {
  UnitTestAll();
  // Return success code
  return 0;
}
```

6 Unit tests output

```
UnitTestJSONCreateFree OK
UnitTestJSONSetGet OK
myStruct:
{
  "_emptyVal":"",
  "_intVal":"1",
  "_escapeVal":"\"double quoted\"",
  "_intArr":["2","3","4"],
  "_emptyArr":"",
  "_oneIntArr":"4",
  "_structVal":{
    "_intVal":"5",
    "_floatVal":"6.000000"
  },
  "_structArr":[
    {
      "_intVal":"7",
      "_floatVal":"8.000000"
    },
    {
      "_intVal":"9",
      "_floatVal":"10.000000"
    }
  ]
}
array:
["8","9","10"]
UnitTestJSONLoadSave OK
UnitTestJSON OK
UnitTestAll OK
```

7 Examples

testJsonReadable.txt:

```
{
  "_emptyVal":"",
  "_intVal":"1",
  "_escapeVal":"\"double quoted\"",
```

```
  "_intArr":["2","3","4"],
  "_emptyArr":"",
  "_oneIntArr":"4",
  "_structVal":{
    "_intVal":"5",
    "_floatVal":"6.000000"
  },
  "_structArr":[
    {
      "_intVal":"7",
      "_floatVal":"8.000000"
    },
    {
      "_intVal":"9",
      "_floatVal":"10.000000"
    }
  ]
}
```


testJsonCompact.txt:


```
{"_emptyVal":"","_intVal":"1","_escapeVal":"\"double quoted\"","_intArr":["2","3","4"],"_emptyArr":"","_oneIntArr":"4
```


testJsonArray.txt:


```
["8","9","10"]
```