tpl User Guide

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
1.5	February 2010		TDH

Contents

1 Overview

1.1 Serialization in C

Tpl is a library for serializing C data. The data is stored in its natural binary form. The API is small and tries to stay "out of the way". Tpl can serialize many C data types, including structures.

1.2 Uses for tpl

Tpl makes a convenient file format. For example, suppose a program needs to store a list of user names and ids. This can be expressed using the format string `A(si)`. If the program needs two such lists (say, one for regular users and one for administrators) this could be expressed as `A(si)A(si)`. It is easy to read and write this kind of structured data using tpl.

Tpl can also be used as an IPC message format. It handles byte order issues and deframing individual messages off of a stream automatically.

1.3 Expressing type

The "data type" of a tpl is explicitly stated as a format string. There is never any ambiguity about the type of data stored in a tpl. Some examples:

- `A(is)` is a variable-length array of integer-string pairs
- `A(is)A(is)` are two such arrays, completely independent of one another
- `S(ci)` is a structure containing a char and integer
- `S(ci)#` is a fixed-length array of the latter structure
- `A(A(i))` is a nested array, that is, an array of integer arrays

1.4 The tpl image

A tpl image is the serialized form of a tpl, stored in a memory buffer or file, or written to a file descriptor.

1.4.1 What's in a tpl image?

There is no need to understand the internal structure of the tpl image. But for the curious, the image is a strictly defined binary buffer having two sections, a header and the data. The header encodes the length of the image, its format string, endian order and other flags. The data section contains the packed data.

1.4.2 No framing needed

A property of the tpl image is that consecutive images can be written to a stream without requiring any delimiter between them. The reader making use of `tpl_gather` (or `tpl_load` in `TPL_FD` mode) will obtain exactly one tpl image at a time. Therefore tpl images can be used as an IPC message format without any higher-level framing protocol.

1.4.3 Data portability

A tpl image generated on one kind of CPU will generally be portable to other CPU types when tpl is used properly. This may be a surprise considering that tpl is a binary format. But tpl has been carefully designed to make this work. Each format character has an associated explicitly-sized type. For integer and floating point types, whose "endian" or byte-order convention varies from one CPU to another, tpl automatically and transparently corrects the endian order (if needed) during the unpacking process. Floating point numbers present their own special difficulties. *No guarantees are made with regard to floating point portability.* That said, because many modern CPU's use IEEE 754 floating point representation, data is likely to be portable among them.

1.5 XML and Perl

Note: The `tplxml` utility and the Perl module are currently unsupported in tpl 1.5.

1.5.1 XML

While a tpl image is a binary entity, you can view any tpl image in XML format using the included `tplxml` utility, located in the `lang/perl` directory.

```
tplxml file.tpl > file.xml
tplxml file.xml > file.tpl
```

The utility is bidirectional, as shown. The file extension is not important; `tplxml` inspects its input to see if it's tpl or XML. You can also pipe data into it instead of giving it a filename. The `tplxml` utility is slow. Its purpose is two-fold: debugging (manual inspection of the data in a tpl), and interoperability with XML-based programs. The resulting XML is often ten times the size of the original binary tpl image.

1.5.2 Perl

There is a Perl module in `lang/perl/Tpl.pm`. The Perl API is convenient for writing Perl scripts that interoperate with C programs, and need to pass structured data back and forth. It is written in pure Perl.

1.6 Platforms

The tpl software was developed for POSIX systems and has been tested on 32- and 64-bit platforms including:

- Linux
- Solaris
- Mac OS X
- OpenBSD
- Windows using Visual Studio 2008 or 2010, or Cygwin or MinGW

1.7 BSD licensed

This software is made available under the revised BSD license. It is free and open source.

1.8 Download

Please follow the link to download on the tpl website.

1.9 Getting help

If you need help, you are welcome to email the author at thanson@users.sourceforge.net.

1.10 Resources

News
: The author has a news feed for software updates (RSS).

2 Build and install

Tpl has no dependencies on libraries other than the system C library. You can simply copy the tpl source into your project, so you have no dependencies. Alternatively, you can build tpl as a library and link it to your program.

2.1 As source

The simplest way to use tpl is to copy the source files `tpl.h` and `tpl.c` (from the `src/` directory) right into your project, and build them with the rest of your source files. No special compiler flags are required.

2.2 As a library

Alternatively, to build tpl as a library, from the top-level directory, run:

```
./configure
make
make install
```

This installs a static library `libtpl.a` and a shared library (e.g., `libtpl.so`), if your system supports them, in standard places. The installation directory can be customized using `./configure --prefix=/some/directory`. Run `configure --help` for further options.

2.2.1 Test suite

You can compile and run the built-in test suite by running:

```
cd tests/
make
```

2.3 On Windows

2.3.1 DLL

On the tpl home page, a Visual Studio 2008 solution package is available for download. This zip file contains pre-built 32- and 64-bit versions of tpl as a DLL. If you like, you can build the DLL yourself using VS2008 or VS2010 (the free Express Edition is sufficient) by opening the solution file and choosing Build Solution.

2.3.2 Non-DLL usage

Alternatively, tpl can be used directly (instead of as a DLL) by compiling the tpl sources right into your program. To do this, add `tpl.c`, `tpl.h`, `win/mman.h` and `win/mmap.c` to your program's source and header files and add the preprocessor definition `TPL_NOLIB`.

2.3.3 MinGW/Cygwin

Prior to tpl release 1.5, using tpl on Windows required building it with MinGW or Cygwin. This is no longer necessary. If you want to build it that way anyway, use the non-Windows (i.e. tar.bz2) tpl download and follow the "configure; make; make install" approach.

3 API concepts

To use tpl, you need to know the order in which to call the API functions, and the background concepts of format string, arrays and index numbers.

3.1 Order of functions

Creating a tpl is always the first step, and freeing it is the last step. In between, you either pack and dump the tpl (if you're serializing data) or you load a tpl image and unpack it (if you're deserializing data).

Table 1: Order of usage

Step	If you're serializing...	If you're deserializing...
1.	`tpl_map()`	`tpl_map()`
2.	`tpl_pack()`	`tpl_load()`
3.	`tpl_dump()`	`tpl_unpack()`
4.	`tpl_free()`	`tpl_free()`

3.2 Format string

When a tpl is created using `tpl_map()`, its data type is expressed as a format string. Each character in the format string has an associated argument of a specific type. For example, this is how a format string and its arguments are passed in to `tpl_map`:

```
tpl_node *tn;
char c;
int i[10];
tn = tpl_map("ci#", &c, i, 10);  /* ci# is our format string */
```

Table 2: Supported format characters

Type	Description	Required argument type
`j`	16-bit signed int	int16_t* or equivalent
`v`	16-bit unsigned int	uint16_t* or equivalent
`i`	32-bit signed int	int32_t* or equivalent
`u`	32-bit unsigned int	uint32_t* or equivalent
`I`	64-bit signed int	int64_t* or equivalent
`U`	64-bit unsigned int	uint64_t* or equivalent
`c`	character (byte)	char*
`s`	string	char**
`f`	64-bit double precision float	double* (varies by platform)
`#`	array length; modifies preceding `iujvIUcsf` or `S(...)`	int
`B`	binary buffer (arbitrary-length)	tpl_bin*
`S`	structure (...)	struct *
`$`	nested structure (...)	none
`A`	array (...)	none

3.2.1 Explicit sizes

The sizes of data types such as `long` and `double` vary by platform. This must be kept in mind because most tpl format characters require a pointer argument to a specific-sized type, listed above. You can use explicit-sized types such as `int32_t` (defined in `inttypes.h`) in your program if you find this helpful.

The trouble with double

Unfortunately there are no standard explicit-sized floating-point types-- no `float64_t`, for example. If you plan to serialize `double` on your platform using tpl's `f` format character, first be sure that your `double` is 64 bits. Second, if you plan to deserialize it on a different kind of CPU, be sure that both CPU's use the same floating-point representation such as IEEE 754.

3.3 Arrays

Arrays come in two kinds: **fixed-length** and **variable-length** arrays. Intuitively, they can be thought of like conventional C arrays and linked lists. In general, use fixed-length arrays if possible, and variable-length arrays if necessary. The variable-length arrays support more complex data types, and give or receive the elements to your program one by one.

3.3.1 Fixed-length vs. Variable-length arrays

Notation
: Fixed-length arrays are denoted like `i#` (a simple type followed by one or more `#` signs), but variable-length arrays are denoted like `A(i)`.

Element handling
: All the elements of a fixed-length array are packed or unpacked at once. But the elements of a variable-length array are packed or unpacked one by one.

Array length
: The number of elements in a fixed-length array is specified before use-- before any data is packed. But variable-length arrays do not have a fixed element count. They can have any number of elements packed into them. When unpacking a variable-length array, they are unpacked one by one until they are exhausted.

Element types
: Elements of fixed-length arrays can be the integer, byte, double, string types or structures. (This excludes format characters `BA`). Fixed-length arrays can also be multi-dimensional like `i##`. Variable-length arrays can have simple or complex elements-- for example, an array of ints `A(i)`, an array of int/double pairs `A(if)`, or even nested arrays like `A(A(if))`.

Before explaining all the concepts, it's illustrative to see how both kinds of arrays are used. Let's pack the integers 0 through 9 both ways.

Packing 0-9 as a fixed-length array

```
#include "tpl.h"
int main() {
    tpl_node *tn;
    int x[] = {0,1,2,3,4,5,6,7,8,9};

    tn = tpl_map("i#", x, 10);
    tpl_pack(tn,0);                          /* pack all 10 elements at once */
    tpl_dump(tn, TPL_FILE, "/tmp/fixed.tpl");
    tpl_free(tn);
}
```

Note that the length of the fixed-length array (10) was passed as an argument to `tpl_map()`. The corresponding unpacking example is listed further below. Now let's see how we would pack 0-9 as a variable-length array:

Packing 0-9 as a variable-length array

```
#include "tpl.h"
int main() {
    tpl_node *tn;
    int x;

    tn = tpl_map("A(i)", &x);
    for(x = 0; x < 10; x++) tpl_pack(tn,1);  /* pack one element at a time */
    tpl_dump(tn, TPL_FILE, "/tmp/variable.tpl");
    tpl_free(tn);
}
```

Notice how we called `tpl_pack` in a loop, once for each element 0-9. Again, there is a corresponding unpacking example shown later in the guide. You might also notice that this time, we passed 1 as the final argument to tpl_pack. This is an index number designating which variable-length array we're packing. In this case, there is only one.

3.3.2 Index numbers

Index numbers identify a particular variable-length array in the format string. Each `A(...)` in a format string has its own index number. The index numbers are assigned left-to-right starting from 1. Examples:

```
A(i)         /* index number 1 */
A(i)A(i)     /* index numbers 1 and 2 */
A(A(i))      /* index numbers 1 and 2 (order is independent of nesting) */
```

Special index number 0

The special index number 0 designates all the format characters that are not inside an `A(...)`. Examples of what index 0 does (and does not) designate:

```
S(ius)       /* index 0 designates the whole thing */
iA(c)u       /* index 0 designates the i and the u */
c#A(i)S(ci) /* index 0 designates the c# and the S(ci) */
```

An index number is passed to `tpl_pack` and `tpl_unpack` to specify which variable-length array (or non-array, in the case of index number 0) to act upon.

3.4 Integers

The array examples above demonstrated how integers could be packed. We'll show some further examples here of unpacking integers and dealing with multi-dimensional arrays. The same program could be used to demonstrate working with byte, 16-bit shorts, 32-bit or 64-bit signed and unsigned integers with only a change to the data type and the format character.

Unpacking 0-9 from a fixed-length array

```
#include "tpl.h"
int main() {
    tpl_node *tn;
    int x[10];

    tn = tpl_map("i#", x, 10);
    tpl_load(tn, TPL_FILE, "/tmp/fixed.tpl");
    tpl_unpack(tn,0);  /* unpack all 10 elements at once */
    tpl_free(tn);
    /* now do something with x[0]...x[9].. (not shown */
}
```

For completeness, let's also see how to unpack a variable-length integer array.

Unpacking 0-9 from a variable-length array

```
#include "tpl.h"
int main() {
    tpl_node *tn;
    int x;

    tn = tpl_map("A(i)", &x);
    tpl_load(tn, TPL_FILE, "/tmp/variable.tpl");
    while (tpl_unpack(tn,1) > 0) printf("%d\n",x); /* unpack one by one */
    tpl_free(tn);
}
```

3.4.1 Multi-dimensional arrays

A multi-dimensional matrix of integers can be packed and unpacked the same way as any fixed-length array.

```
int xy[XDIM][YDIM];
...
tn = tpl_map("i##", xy, XDIM, YDIM);
tpl_pack(tn, 0);
```

This single call to `tpl_pack` packs the entire matrix.

3.5 Strings

Tpl can serialize C strings. A different format is used for `char*` vs. `char[ ]` as described below. Let's look at `char*` first:

Packing a string

```
    #include "tpl.h"

    int main() {
        tpl_node *tn;
        char *s = "hello, world!";
        tn = tpl_map("s", &s);
        tpl_pack(tn,0);  /* copies "hello, world!" into the tpl */
        tpl_dump(tn,TPL_FILE,"string.tpl");
        tpl_free(tn);
    }
```

The `char*` must point to a null-terminated string or be a `NULL` pointer.

When deserializing (unpacking) a C string, space for it will be allocated automatically, but you are responsible for freeing it (unless it is `NULL`):

Unpacking a string

```
    #include "tpl.h"

    int main() {
        tpl_node *tn;
        char *s;
        tn = tpl_map("s", &s);
        tpl_load(tn,TPL_FILE,"string.tpl");
        tpl_unpack(tn,0);   /* allocates space, points s to "hello, world!" */
        printf("unpacked %s\n", s);
        free(s);            /* our responsibility to free s */
        tpl_free(tn);
    }
```

3.5.1 char* vs char[ ]

The `s` format character is only for use with `char*` types. In the example above, `s` is a `char*`. If it had been a `char s[14]`, we would use the format characters `c#` to pack or unpack it, as a fixed-length character array. (This unpacks the characters "in-place", instead of into a dynamically allocated buffer). Also, a fixed-length buffer described by `c#` need not be null-terminated.

3.5.2 Arrays of strings

You can use fixed- or variable-length arrays of strings in tpl. An example of packing a fixed-length two-dimensional array of strings is shown here.

```
char *labels[2][3] = { {"one", "two", "three"},
                       {"eins", "zwei", "drei" } };
tpl_node *tn;
tn = tpl_map("s##", labels, 2, 3);
tpl_pack(tn,0);
tpl_dump(tn,TPL_FILE,filename);
tpl_free(tn);
```

Later, when unpacking these strings, the programmer must remember to free them one by one, after they are no longer needed.

```
char *olabels[2][3];
int i,j;

tn = tpl_map("s##", olabels, 2, 3);
tpl_load(tn,TPL_FILE,filename);
tpl_unpack(tn,0);
tpl_free(tn);

for(i=0;i<2;i++) {
  for(j=0;j<3;j++) {
    printf("%s\n", olabels[i][j]);
    free(olabels[i][j]);
  }
}
```

3.6 Binary buffers

Packing an arbitrary-length binary buffer (tpl format character `B`) makes use of the `tpl_bin` structure. You must declare this structure and populate it with the address and length of the binary buffer to be packed.

Packing a binary buffer

```
    #include "tpl.h"
    #include <sys/time.h>

    int main() {
        tpl_node *tn;
        tpl_bin tb;

        /* we'll use a timeval as our guinea pig */
        struct timeval tv;
        gettimeofday(&tv,NULL);

        tn = tpl_map( "B", &tb );
        tb.sz = sizeof(struct timeval);  /* size of buffer to pack */
```

```
        tb.addr = &tv;                  /* address of buffer to pack */
        tpl_pack( tn, 0 );
        tpl_dump(tn, TPL_FILE, "bin.tpl");
        tpl_free(tn);
    }
```

When you unpack a binary buffer, tpl will automatically allocate it, and will populate your `tpl_bin` structure with its address and length. You are responsible for eventually freeing the buffer.

Unpacking a binary buffer

```
    #include "tpl.h"

    int main() {
        tpl_node *tn;
        tpl_bin tb;

        tn = tpl_map( "B", &tb );
        tpl_load( tn, TPL_FILE, "bin.tpl" );
        tpl_unpack( tn, 0 );
        tpl_free(tn);

        printf("binary buffer of length %d at address %p\n", tb.sz, tb.addr);
        free(tb.addr);  /* our responsibility to free it */
    }
```

3.7 Structures

You can use tpl to pack and unpack structures, and arrays of structures.

```
struct ci {
    char c;
    int i;
};
struct ci s = {'a', 1};

tn = tpl_map("S(ci)", &s);  /* pass structure address */
tpl_pack(tn, 0);
tpl_dump(tn, TPL_FILE, "struct.tpl");
tpl_free(tn);
```

As shown, omit the individual arguments for the format characters inside the parenthesis. The exception is for fixed-length arrays; when `S(...)` contains a `#` character, its length argument is required: `tpl_map("S(f#i)", &s, 10);`

When using the `S(...)` format, the only characters allowed inside the parentheses are `iujvcsfIU#$()`.

3.7.1 Structure arrays

Arrays of structures are the same as simple arrays. Fixed- or variable- length arrays are supported.

```
struct ci sa[100], one;

tn = tpl_map("S(ci)#", sa, 100);  /* fixed-length array of 100 structures */
tn = tpl_map("A(S(ci))", &one);   /* variable-length array (one at a time)*/
```

The differences between fixed- and variable-length arrays are explained in the Arrays section.

3.7.2 Nested structures

When dealing with nested structures, the outermost structure uses the `S` format character, and the inner nested structures use the `$` format. Only the *outermost* structure's address is given to `tpl_map`.

```
struct inner_t {
  char a;
}

struct outer_t {
  char b;
  struct inner_t i;
}

tpl_node *tn;
struct outer_t outer = {'b', {'a'}};

tn = tpl_map("S(c$(c))", &outer);
```

Structures can nest to any level. Currently tpl does not support fixed-length array suffixes on inner structures. However the outermost structure can have a length suffix even if it contains some nested structures.

3.8 Linked lists

While tpl has no specific data type for a linked list, the technique for packing them is illustrated here. First describe your list element as a format string and then surround it with `A(...)` to describe it as variable-length array. Then, using a temporary variable, iterate over each list element, copying it to the temporary variable and packing it.

```
struct element {
  char c;
  int i;
  struct element *next;
}

struct element *list, *i, tmp;
tpl_node *tn;

/* add some elements to list.. (not shown)*/

tn = tpl_map("A(ci)", &tmp);
for(i = list; i != NULL; i=i->next) {
  tmp = *i;
  tpl_pack(tn, 1);
}
tpl_dump(tn,TPL_FILE,"list.tpl");
tpl_free(tn);
```

Unpacking is similar. The `for` loop is just replaced with:

```
while( tpl_unpack(tn,1) > 0) {
  struct element *newelt = malloc(sizeof(struct element));
  *newelt = tmp;
  add_to_list(list, newelt);
}
```

As you can see, tpl does not reinstate the whole list at once-- just one element at a time. You need to link the elements manually. A future release of tpl may support *pointer swizzling* to make this easier.

4 API

4.1 tpl_map

The only way to create a tpl is to call `tpl_map()`. The first argument is the format string. This is followed by a list of arguments as required by the particular characters in the format string. E.g,

```
tpl_node *tn;
int i;
tn = tpl_map( "A(i)", &i );
```

The function creates a mapping between the items in the format string and the C program variables whose addresses are given. Later, the C variables will be read or written as the tpl is packed or unpacked.

This function returns a `tpl_node*` on success, or `NULL` on failure.

4.2 tpl_pack

The function `tpl_pack()` packs data into a tpl. The arguments to `tpl_pack()` are a `tpl_node*` and an index number.

```
tn = tpl_map("A(i)A(c)", &i, &c);
for(i=0; i<10; i++) tpl_pack(tn, 1);     /* pack 0-9 into index 1 */
for(c='a'; c<='z'; c++) tpl_pack(tn, 2); /* pack a-z into index 2 */
```

> **Data is copied when packed**
>
> Every call to `tpl_pack()` immediately *copies* the data being packed. Thus the program is free to immediately overwrite or re-use the packed variables.

4.2.1 Index number 0

It is necessary to pack index number 0 only if the format string contains characters that are not inside an `A(...)`, such as the `i` in the format string `iA(c)`.

4.2.2 Variable-length arrays

Adding elements to an array

To add elements to a variable-length array, call `tpl_pack()` repeatedly. Each call adds another element to the array.

Zero-length arrays are ok

It's perfectly acceptable to pack nothing into a variable-length array, resulting in a zero-length array.

Packing nested arrays

In a format string containing a nested, variable-length array, such as `A(A(s))`, the inner, child array should be packed prior to the parent array.

When you pack a parent array, a "snapshot" of the current child array is placed into the parent's new element. Packing a parent array also empties the child array. This way, you can pack new data into the child, then pack the parent again. This creates distinct parent elements which each contain distinct child arrays.

Tip
When dealing with nested arrays like `A(A(i))`, *pack* them from the "inside out" (child first), but *unpack* them from the "outside in" (parent first).

The example below creates a tpl having the format string `A(A(c))`.

Packing nested arrays

```
#include "tpl.h"

int main() {
    char c;
    tpl_node *tn;

    tn = tpl_map("A(A(c))", &c);

    for(c='a'; c<'c'; c++) tpl_pack(tn,2);  /* pack child (twice) */
    tpl_pack(tn, 1);                        /* pack parent */

    for(c='1'; c<'4'; c++) tpl_pack(tn,2);  /* pack child (three times) */
    tpl_pack(tn, 1);                        /* pack parent */

    tpl_dump(tn, TPL_FILE, "test40.tpl");
    tpl_free(tn);
}
```

This creates a nested array in which the parent has two elements: the first element is the two-element nested array *a*, *b*; and the second element is the three-element nested array *1*, *2*, *3*. The nested unpacking example shows how this tpl is unpacked.

4.3 tpl_dump

After packing a tpl, `tpl_dump()` is used to write the tpl image to a file, memory buffer or file descriptor. The corresponding modes are shown below. A final mode is for querying the output size without actually performing the dump.

Write to…	Usage
file	`tpl_dump(tn, TPL_FILE, "file.tpl" );`
file descriptor	`tpl_dump(tn, TPL_FD, 2);`
memory	`tpl_dump(tn, TPL_MEM, &addr, &len );`
caller's memory	`tpl_dump(tn, TPL_MEM|TPL_PREALLOCD, buf, sizeof(buf));`
just get size	`tpl_dump(tn, TPL_GETSIZE, &sz);`

The first argument is the `tpl_node*` and the second is one of these constants:

`TPL_FILE`
Writes the tpl to a file whose name is given in the following argument. The file is created with permissions 664 (`rw-rw-r--`) unless further restricted by the process `umask`.

`TPL_FD`
Writes the tpl to the file descriptor given in the following argument. The descriptor can be either blocking or non-blocking, but will busy-loop if non-blocking and the contents cannot be written immediately.

`TPL_MEM`
Writes the tpl to a memory buffer. The following two arguments must be a `void\*\*` and a `size_t*`. The function will allocate a buffer and store its address and length into these locations. The caller is responsible to `free()` the buffer when done using it.

`TPL_MEM|TPL_PREALLOCD`
Writes the tpl to a memory buffer that the caller has already allocated or declared. The following two arguments must be

a `void*` and a `size_t` specifying the buffer address and size respectively. (If the buffer is of insufficient size to receive the tpl dump, the function will return -1). This mode can be useful in conjunction with `tpl_load` in `TPL_EXCESS_OK` mode, as shown here.

`TPL_GETSIZE`
This special mode does not actually dump the tpl. Instead it places the size that the dump *would* require into the `size_t` pointed to by the following argument.

The return value is 0 on success, or -1 on error.

The `tpl_dump()` function does not free the tpl. Use `tpl_free()` to release the tpl's resources when done.

Tip
If you want to store a series of tpl images, or transmit sequential tpl images over a socket (perhaps as messages to another program), you can simply dump them sequentially without needing to add any delimiter for the individual tpl images. Tpl images are internally delimited, so `tpl_load` will read just one at a time even if multiple images are contiguous.

4.4 tpl_load

This API function reads a previously-dumped tpl image from a file, memory buffer or file descriptor, and prepares it for subsequent unpacking. The format string specified in the preceding call to `tpl_map()` will be cross-checked for equality with the format string stored in the tpl image.

```
tn = tpl_map( "A(i)", &i );
tpl_load( tn, TPL_FILE, "demo.tpl" );
```

The first argument to `tpl_load()` is the `tpl_node*`. The second argument is one of the constants:

`TPL_FILE`
Loads the tpl from the file named in the following argument. It is also possible to bitwise-OR this flag with `TPL_EXCESS_OK` as explained below.

`TPL_MEM`
Loads the tpl from a memory buffer. The following two arguments must be a `void*` and a `size_t`, specifying the buffer address and size, respectively. The caller must not free the memory buffer until after freeing the tpl with `tpl_free()`. (If the caller wishes to hand over responsibility for freeing the memory buffer, so that it's automatically freed along with the tpl when `tpl_free()` is called, the constant `TPL_UFREE` may be bitwise-OR'd with `TPL_MEM` to achieve this). Furthermore, `TPL_MEM` may be bitwise-OR'd with `TPL_EXCESS_OK`, explained below.

`TPL_FD`
Loads the tpl from the file descriptor given in the following argument. The descriptor is read until one complete tpl image is loaded; no bytes past the end of the tpl image will be read. The descriptor can be either blocking or non-blocking, but will busy-loop if non-blocking and the contents cannot be read immediately.

During loading, the tpl image will be extensively checked for internal validity.

This function returns 0 on success or -1 on error.

4.4.1 TPL_EXCESS_OK

When reading a tpl image from a file or memory (but not from a file descriptor) the size of the file or memory buffer must exactly equal that of the tpl image stored therein. In other words, no excess trailing data beyond the tpl image is permitted. The bit flag `TPL_EXCESS_OK` can be OR'd with `TPL_MEM` or `TPL_FILE` to relax this requirement.

A situation where this flag can be useful is in conjunction with `tpl_dump` in the `TPL_MEM|TPL_PREALLOCD` mode. In this example, the program does not concern itself with the actual tpl size as long as `LEN` is sufficiently large.

```
char buf[LEN];  /* will store and read tpl images here */
...
tpl_dump(tn, TPL_MEM|TPL_PREALLOCD, buf, LEN);
...
tpl_load(tn, TPL_MEM|TPL_EXCESS_OK, buf, LEN);
```

4.5 tpl_unpack

The `tpl_unpack()` function unpacks data from the tpl. When data is unpacked, it is copied to the C program variables originally specified in `tpl_map()`. The first argument to `tpl_unpack` is the `tpl_node*` for the tpl and the second argument is an index number.

```
tn = tpl_map( "A(i)A(c)", &i, &c );
tpl_load( tn, TPL_FILE, "nested.tpl" );
while (tpl_unpack( tn, 1) > 0) printf("i is %d\n", i); /* unpack index 1 */
while (tpl_unpack( tn, 2) > 0) printf("c is %c\n", c); /* unpack index 2 */
```

4.5.1 Index number 0

It is necessary to unpack index number 0 only if the format string contains characters that are not inside an `A(...)`, such as the `i` in the format string `iA(c)`.

4.5.2 Variable-length arrays

Unpacking elements from an array

For variable-length arrays, each call to `tpl_unpack()` unpacks another element. The return value can be used to tell when you're done: if it's positive, an element was unpacked; if it's 0, nothing was unpacked because there are no more elements. A negative retun value indicates an error (e.g. invalid index number). In this document, we usually unpack variable-length arrays using a `while` loop:

```
while( tpl_unpack( tn, 1 ) > 0 ) {
    /* got another element */
}
```

Array length

When unpacking a variable-length array, it may be convenient to know ahead of time how many elements will need to be unpacked. You can use `tpl_Alen()` to get this number.

Unpacking nested arrays

In a format string containing a nested variable-length array such as `A(A(s))`, unpack the outer, parent array before unpacking the child array.

When you unpack a parent array, it prepares the child array for unpacking. After unpacking the elements of the child array, the program can repeat the process by unpacking another parent element, then the child elements, and so on. The example below unpacks a tpl having the format string `A(A(c))`.

Unpacking nested arrays

```
#include "tpl.h"
#include <stdio.h>

int main() {
```

```
    char c;
    tpl_node *tn;

    tn = tpl_map("A(A(c))", &c);

    tpl_load(tn, TPL_FILE, "test40.tpl");
    while (tpl_unpack(tn,1) > 0) {
        while (tpl_unpack(tn,2) > 0) printf("%c ",c);
        printf("\n");
    }
    tpl_free(tn);
}
```

The file `test40.tpl` is from the nested packing example. When run, this program prints:

```
a b
1 2 3
```

4.6 tpl_free

The final step for any tpl is to release it using `tpl_free()`. Its only argument is the the `tpl_node*` to free.

```
tpl_free( tn );
```

This function does not return a value (it is `void`).

4.7 tpl_Alen

This function takes a `tpl_node*` and an index number and returns an `int` specifying the number of elements in the variable-length array.

```
num_elements = tpl_Alen(tn, index);
```

This is mainly useful for programs that unpack data and need to know ahead of time the number of elements that will need to be unpacked. (It returns the current number of elements; it will decrease as elements are unpacked).

4.8 tpl_peek

This function peeks into a file or a memory buffer containing a tpl image and and returns a copy of its format string. It can also peek at the lengths of any fixed-length arrays in the format string, or it can also peek into the data stored in the tpl.

4.8.1 Format peek

The format string can be obtained like this:

```
fmt = tpl_peek(TPL_FILE, "file.tpl");
fmt = tpl_peek(TPL_MEM, addr, sz);
```

On success, a copy of the format string is returned. The caller must eventually free it. On error, such as a non-existent file, or an invalid tpl image, it returns `NULL`.

4.8.2 Array length peek

The lengths of all fixed-length arrays in the format string can be queried using the `TPL_FXLENS` mode. It provides the number of such fixed-length arrays and their lengths. If the former is non-zero, the caller must free the latter array when finished. The format string itself must also be freed.

```
uint32_t num_fxlens, *fxlens, j;
fmt = tpl_peek(TPL_FILE|TPL_FXLENS, filename, &num_fxlens, &fxlens);
if (fmt) {
  printf("format %s, num_fxlens %u\n", fmt, num_fxlens);
  for(j=0; j<num_fxlens; j++) printf("fxlens[%u] %u\n", j, fxlens[j]);
  if (num_fxlens > 0) free(fxlens);
  free(fmt);
}
```

The `TPL_FXLENS` mode is mutually exclusive with `TPL_DATAPEEK`.

4.8.3 Data peek

To peek into the data, additional arguments are used. This is a quick alternative to mapping, loading and unpacking the tpl, but peeking is limited to the data in index 0. In other words, no peeking into `A(...)` types. Suppose the tpl image in `file.tpl` has the format string `siA(i)`. Then the index 0 format characters are `si`. This is how to peek at their content:

```
char *s;
int i;
fmt = tpl_peek(TPL_FILE | TPL_DATAPEEK, "file.tpl", "si", &s, &i);
```

Now `s`, `i`, and `fmt` have been populated with data. The caller must eventually free `fmt` and `s` because they are allocated strings. Of course, it works with `TPL_MEM` as well as `TPL_FILE`. Notice that `TPL_DATAPEEK` was OR'd with the mode. You can also specify *any leading portion* of the index 0 format if you don't want to peek at the whole thing:

```
fmt = tpl_peek(TPL_FILE | TPL_DATAPEEK, "file.tpl", "s", &s);
```

The `TPL_DATAPEEK` mode is mutually exclusive with `TPL_FXLENS`.

Structure peek

Lastly you can peek into `S(...)` structures in index 0, but omit the surrounding `S(...)` in the format, and specify an argument to receive each structure member individually. You can specify any leading portion of the structure format. For example if `struct.tpl` has the format string `S(si)`, you can peek at its data in these ways:

```
fmt = tpl_peek(TPL_FILE | TPL_DATAPEEK, "struct.tpl", "s", &s);
fmt = tpl_peek(TPL_FILE | TPL_DATAPEEK, "struct.tpl", "si", &s, &i);
```

4.9 tpl_jot

This is a quick shortcut for generating a tpl. It can be used instead of the usual "map, pack, dump, and free" lifecycle. With `tpl_jot` all those steps are handled for you. It only works for simple formats-- namely, those without `A(...)` in their format string. Here is how it is used:

```
char *hello = "hello", *world = "world";
tpl_jot( TPL_FILE, "file.tpl", "ss", &hello, &world);
```

It supports the three standard modes, `TPL_FILE`, `TPL_FD` and `TPL_MEM`. It returns -1 on failure (such as a bad format string or error writing the file) or 0 on success.

4.10 tpl_hook

Most users will just leave these hooks at their default values. You can change these hook values if you want to modify tpl's internal memory management and error reporting behavior.

A global structure called `tpl_hook` encapsulates the hooks. A program can reconfigure any hook by specifying an alternative function whose prototype matches the default. For example:

```
#include "tpl.h"
extern tpl_hook_t tpl_hook;

int main() {
    tpl_hook.oops = printf;
    ...
}
```

Table 3: Configurable hooks

Hook	Description	Default
`tpl_hook.oops`	log error messages	`tpl_oops`
`tpl_hook.malloc`	allocate memory	`malloc`
`tpl_hook.realloc`	reallocate memory	`realloc`
`tpl_hook.free`	free memory	`free`
`tpl_hook.fatal`	log fatal message and exit	`tpl_fatal`
`tpl_hook.gather_max`	tpl_gather max image size	`0 (unlimited)`

4.10.1 The oops hook

The `oops` has the same prototype as `printf`. The built-in default oops handling function writes the error message to `stderr`.

4.10.2 The fatal hook

The fatal hook is invoked when a tpl function cannot continue because of an out- of-memory condition or some other usage violation or inconsistency. It has this prototype:

```
void fatal_fcn(char *fmt, ...);
```

The `fatal` hook must not return. It must either exit, *or* if the program needs to handle the failure and keep executing, `setjmp` and `longjmp` can be used. The default behavior is to `exit(-1)`.

Using longjmp in a fatal error handler

```
#include <setjmp.h>
#include <stdio.h>
#include <stdarg.h>
#include "tpl.h"

jmp_buf env;
extern tpl_hook_t tpl_hook;

void catch_fatal(char *fmt, ...) {
  va_list ap;

  va_start(ap, fmt);
```

```
  vfprintf(stderr, fmt, ap);
  va_end(ap);
  longjmp(env,-1);                /* return to setjmp point */
}

int main() {
  int err;
  tpl_node *tn;
  tpl_hook.fatal = catch_fatal;    /* install fatal handler */

  err = setjmp(env); /* on error, control will return here  */
  if (err) {
    printf("caught error!\n");
    return -1;
  }

  tn = tpl_map("@");              /* generate a fatal error */
  printf("program ending, without error\n");
  return 0;
}
```

This example is included in `tests/test123.c`. When run, this program prints:

```
unsupported option @
failed to parse @
caught error!
```

4.11 tpl_gather

Most programs don't need this

Normally, `tpl_load()` is used to read a tpl image having an expected format string. A more generic operation is to acquire a tpl image whose format string is unknown. E.g., a generic message-receiving function might gather tpl images of varying format and route them to their final destination. This is the purpose of `tpl_gather`. It produces a memory buffer containing one tpl image. If there are multiple contiguous images in the input, it gathers exactly one image at a time.

The prototype for this function is:

```
int tpl_gather( int mode, ...);
```

The `mode` argument is one of three constants listed below, which must be followed by the mode-specific required arguments:

```
TPL_GATHER_BLOCKING,    int fd, void **img, size_t *sz
TPL_GATHER_NONBLOCKING, int fd, tpl_gather_t **gs, tpl_gather_cb *cb, void *data
TPL_GATHER_MEM,         void *addr, size_t sz, tpl_gather_t **gs, tpl_gather_cb *cb, void
```

Note

All modes honor `tpl_hook.gather_max`, specifying the maximum byte size for a tpl image to be gathered (the default is unlimited, signified by 0). If a source attempts to send a tpl image larger than this maximum, whatever partial image has been read will be discarded, and no further reading will take place; in this case `tpl_gather` will return a negative (error) value to inform the caller that it should stop gathering from this source, and close the originating file descriptor if there is one. (The whole idea is to prevent untrusted sources from sending extremely large tpl images which would consume too much memory.)

4.11.1 TPL_GATHER_BLOCKING

In this mode, `tpl_gather` blocks while reading file descriptor `fd` until one complete tpl image is read. No bytes past the end of the tpl image will be read. The address of the buffer containing the image is returned in `img` and its size is placed in `sz`. The caller is responsible for eventually freeing the buffer. The function returns 1 on success, 0 on end-of-file, or a negative number on error.

4.11.2 TPL_GATHER_NONBLOCKING

This mode is for non-blocking, event-driven programs that implement their own file descriptor readability testing using `select()` or the like. In this mode, tpl images are gathered in chunks as data becomes readable. Whenever a full tpl image has been gathered, it invokes a caller-specified callback to do something with the image. The arguments are the file descriptor `fd` which the caller has determined to be readable and which must be in non-blocking mode, a pointer to a file-descriptor-specific handle which the caller has declared (explained below); a callback to invoke when a tpl image has been read; and an opaque pointer that will passed to the callback.

For each file descriptor on which `tpl_gather` will be used, the caller must declare a `tpl_gather_t*` and initialize it to `NULL`. Thereafter it will be used internally by `tpl_gather` whenever data is readable on the descriptor.

The callback will only be invoked whenever `tpl_gather()` has accumulated one complete tpl image. It must have this prototype:

```
int (tpl_gather_cb)(void *img, size_t sz, void *data);
```

The callback can do anything with the tpl image but it must not free it. It can be copied if it needs to survive past the callback's return. The callback should return 0 under normal circumstances, or a negative number to abort; that is, returning a negative number causes `tpl_gather` itself to discard any remaining full or partial tpl images that have been read, and to return a negative number (-4 in particular) to signal its caller to close the file descriptor.

The return value of `tpl_gather()` is negative if an error occured or 0 if a normal EOF was encountered-- both cases require that the caller close the file descriptor (and stop monitoring it for readability, obviously). If the return value is positive, the function succeeded in gathering whatever data was currently readable, which may have been a partial tpl image, or one or more complete images.

Typical Usage

The program will have established a file descriptor in non-blocking mode and be monitoring it for readability, using `select()`. Whenever it's readable, the program calls `tpl_gather()`. In skeletal terms:

```
tpl_gather_t *gt=NULL;
int rc;

void fd_is_readable(int fd) {
  rc = tpl_gather( TPL_GATHER_NONBLOCKING, fd, &gt, callback, NULL );
  if (rc <= 0) {
      close(fd);               /* got eof or fatal */
      stop_watching_fd(fd);
  }
}

int callback( void *img, size_t sz, void *data ) {
  printf("got a tpl image\n"); /* do something with img. do not free it. */
  return 0;                    /* normal (no error) */
}
```

4.11.3 TPL_GATHER_MEM

This mode is identical to `TPL_GATHER_NONBLOCKING` except that it gathers from a memory buffer instead of from a file descriptor. In other words, if some other layer of code-- say, a decryption function (that is decrypting fixed-size blocks) produces tpl fragments one-by-one, this mode can be used to reconstitute the tpl images and invoke the callback for each one. Its parameters are the same as for the `TPL_GATHER_NONBLOCKING` mode except that instead of a file descriptor, it takes a buffer address and size. The return values are also the same as for `TPL_GATHER_NONBLOCKING` noting of course there is no file descriptor to close on a non-positive return value.