StrPack.jl Documentation

Release development

Patrick O'Leary

Dec 09, 2017

Contents

This package performs serialization of structures to and deserialization from binary data streams and string objects.

CHAPTER 1

Example

Let's create a C library as follows:

```
struct teststruct {
  int      int1;
  float    float1;
};

void getvalues(struct teststruct* ts)
{
  ts->int1 = 7;
  ts->float1 = 3.7;
}
```

The `getvalues` function just fills the two fields with specified values. Compile this as a shared library, which on Linux is achieved with `gcc -fPIC teststruct.c -shared -o libteststruct.so`.

Let's also create the Julia analog of this structure:

```
using StrPack

@struct type TestStruct
    int1::Int32
    float1::Float32
end
#define a convenience constructor for 0.2 versions of Julia (in 0.3 this is automatic)
if VERSION < v"0.3.0-pre"
  TestStruct(i, f) = TestStruct(convert(Int32, i), convert(Float32, f))
end
```

Note that C's `int` corresponds to `Int32`. Let's initialize an object of this type:

```
s = TestStruct(-1, 1.2)
```

We can pack `s` into a form suitable to pass as the input to our C function `getvalues`, which we do in the following way:

```
iostr = IOBuffer()
pack(iostr, s)
```

It's worth seeing what has happened here:

```
julia> iostr
IOBuffer([0xff, 0xff, 0xff, 0xff, 0x9a, 0x99, 0x99, 0x3f],9)
```

The first 4 bytes correspond to the `Int32` representation of -1, and the last 4 to the `Float32` representation of 1.2. In other words, this is just a packed memory buffer encoding `s`. (There are subtleties such as data alignment, endian status, etc. strpack knows about this stuff, and users who need to control its behavior manually can do so.)

Now we load our library and make a `ccall`:

```
const libtest = dlopen("libteststruct")
ccall(dlsym(libtest, :getvalues), Void, (Ptr{Void},), iostr.data)
```

The C function `getvalues` stores its output in the buffer we provided as input. We unpack this buffer back into a Julia type:

```
seek(iostr, 0)    # "rewind" to the beginning of the buffer
s2 = unpack(iostr, TestStruct)
```

Voila! You have the result back.

CHAPTER 2

Macros

@struct (*type*, *strategy*, *endianness*)

Create and register a structural Julia type with StrPack. The type argument uses an extended form of the standard Julia type syntax to define the size of arrays and strings. Each element must declare its type, and each type must be reducible to a bits type or array or composite of bits types.:

```
@struct type StructuralType
    a::Float64 # a bits type
    b::Array{Int32,2}(4, 4) # an array of bits types
    c::ASCIIString(8) # a string with a fixed number of bytes
end
```

CHAPTER 3

Methods

pack(*io*, *composite*[, *asize*, *strategy*, *endianness*])

Create a packed buffer representation of `composite` in stream `io`, using array and string sizes fixed by `asize` and data alignment coded by `strategy` with endianness `endianness`. If the optional arguments are not provided, then `T`, the type of `composite`, is expected to have been created with the `@struct` macro.

unpack(*io*, *T*[, *asize*, *strategy*, *endianness*])

Extract an instance of the Julia composite type `T` from the packed representation in the stream `io`. If the optional arguments are not provided, then `T` is expected to have been created with the `@struct` macro.

show_struct_layout(*T[, asize, strategy][, width, bytesize]*)

Print a graphical representation of the memory layout of the packed type `T`. If `asize` and `strategy` are not provided, then `T` is expected to have been created with the `@struct` macro. The display will show `width` bytes in each row, with each byte taking up `bytesize` characters.

CHAPTER 4

Endianness

StrPack supports both big-endian (also known as network-ordered) and little-endian streams. The symbols `:BigEndian`, `:LittleEndian`, and `:NativeEndian` can be passed as `endianness` arguments in StrPack macros and methods.

CHAPTER 5

Packing strategies

To support arbitrary ABIs, StrPack defines a number of "packing strategies"–that is, instructions for inserting padding bytes–as well as allowing the user to define their own.

The predefined strategies are:

`align_default` Each bits type is aligned to the next higher power of two. Arrays and structures are aligned to the largest alignment required by any of their members.

`align_packed` No padding is inserted. Equivalent to `__attribute__ (( __packed__ ))`.

`align_x86_pc_linux_gnu` The x86 Linux ABI, which uses 4 byte alignment for `Int64`, `Uint64`, and `Float64`.

`align_native` The native C ABI specified by the host platform.

`align_packmax(n)` The default alignment is used up to a limit of `n` bytes. Equvalent to `#pragma pack(n)`.

`align_structpack(n)` The default alignment is used for bitstypes, but aggregate types are aligned to `n` bytes. Equivalent to `__attribute__ (( align(n) ))`

`align_table(ttable)` The alignments will be taken from `ttable`, a `Dict` mapping types to alignments in bytes. Otherwise, the default alignment will be used.

The `align_packmax`, `align_structpack`, and `align_table` strategies can be composed. For example:

```
align_structpack(align_table({Special => 2}), 4)
```

will align structures to 4 bytes, except for `Special`, which will be aligned to 2 bytes.

Completely custom alignment strategies can be defined by constructing a `DataAlign` type.

`DataAlign`(*[ttable]*, *default*, *aggregate*)
Create a padding strategy. `ttable` is an optional `Dict` mapping types to alignments in bytes. `default` is a function from `Type` to `Integer` which should return the required alignment for bitstypes not in `ttable`. `aggregate` is a function from `Vector{Type}` to `Integer` which should return the requirement for composite (also known as structure or aggreggate) types.

Index