Convert::Binary::C

Binary Data Conversion using C Types

Version 0.69

Marcus Holland-Moritz
mhx@cpan.org

December 6, 2007

Contents

Convert::Binary::C

Binary Data Conversion using C Types

1.1 Synopsis

1.1.1 Simple

```
use Convert::Binary::C;

#-------------------------------------------
# Create a new object and parse embedded code
#-------------------------------------------
my $c = Convert::Binary::C->new->parse(<<ENDC);

enum Month { JAN, FEB, MAR, APR, MAY, JUN,
             JUL, AUG, SEP, OCT, NOV, DEC };

struct Date {
  int        year;
  enum Month month;
  int        day;
};

ENDC

#-----------------------------------------------
# Pack Perl data structure into a binary string
#-----------------------------------------------
my $date = { year => 2002, month => 'DEC', day => 24 };

my $packed = $c->pack('Date', $date);
```

1.1.2 Advanced

```
use Convert::Binary::C;
use Data::Dumper;

#---------------------
# Create a new object
#---------------------
my $c = new Convert::Binary::C ByteOrder => 'BigEndian';
```

```
#------------------------------------------------
# Add include paths and global preprocessor defines
#------------------------------------------------
$c->Include('/usr/lib/gcc-lib/i686-pc-linux-gnu/3.3.6/include',
            '/usr/include')
  ->Define(qw( __USE_POSIX __USE_ISOC99=1 ));

#---------------------------------
# Parse the 'time.h' header file
#---------------------------------
$c->parse_file('time.h');

#---------------------------------------
# See which files the object depends on
#---------------------------------------
print Dumper([$c->dependencies]);

#-------------------------------------------------------------
# See if struct timespec is defined and dump its definition
#-------------------------------------------------------------
if ($c->def('struct timespec')) {
  print Dumper($c->struct('timespec'));
}

#-------------------------------
# Create some binary dummy data
#-------------------------------
my $data = "binary_test_string";

#-----------------------------------------------------
# Unpack $data according to 'struct timespec' definition
#-----------------------------------------------------
if (length($data) >= $c->sizeof('timespec')) {
  my $perl = $c->unpack('timespec', $data);
  print Dumper($perl);
}

#-----------------------------------------------------
# See which member lies at offset 5 of 'struct timespec'
#-----------------------------------------------------
my $member = $c->member('timespec', 5);
print "member('timespec', 5) = '$member'\n";
```

1.2 Description

Convert::Binary::C is a preprocessor and parser for C type definitions. It is highly configurable and supports arbitrarily complex data structures. Its object-oriented interface has `pack` and `unpack` methods that act as replacements for Perl's `pack` and `unpack` and allow to use C types instead of a string representation of the data structure for conversion of binary data from and to Perl's complex data structures.

Actually, what Convert::Binary::C does is not very different from what a C compiler does, just that it doesn't compile the source code into an object file or executable, but only parses the code and allows Perl to use the enumerations, structs, unions and typedefs that have been defined within your C source for binary data conversion, similar to Perl's `pack` and `unpack`.

Beyond that, the module offers a lot of convenience methods to retrieve information about the C types that have been parsed.

1.2.1 Background and History

In late 2000 I wrote a real-time debugging interface for an embedded medical device that allowed me to send out data from that device over its integrated Ethernet adapter. The interface was `printf()`-like, so you could easily send out strings or numbers. But you could also send out what I called *arbitrary data*, which was intended for arbitrary blocks of the device's memory.

Another part of this real-time debugger was a Perl application running on my workstation that gathered all the messages that were sent out from the embedded device. It printed all the strings and numbers, and hex-dumped the arbitrary data. However, manually parsing a couple of 300 byte hex-dumps of a complex C structure is not only frustrating, but also error-prone and time consuming.

Using `unpack` to retrieve the contents of a C structure works fine for small structures and if you don't have to deal with struct member alignment. But otherwise, maintaining such code can be as awful as deciphering hex-dumps.

As I didn't find anything to solve my problem on the CPAN, I wrote a little module that translated simple C structs into `unpack` strings. It worked, but it was slow. And since it couldn't deal with struct member alignment, I soon found myself adding padding bytes everywhere. So again, I had to maintain two sources, and changing one of them forced me to touch the other one.

All in all, this little module seemed to make my task a bit easier, but it was far from being what I was thinking of:

- A module that could directly use the source I've been coding for the embedded device without any modifications.
- A module that could be configured to match the properties of the different compilers and target platforms I was using.
- A module that was fast enough to decode a great amount of binary data even on my slow workstation.

I didn't know how to accomplish these tasks until I read something about XS. At least, it seemed as if it could solve my performance problems. However, writing a C parser in C isn't easier than it is in Perl. But writing a C preprocessor from scratch is even worse.

Fortunately enough, after a few weeks of searching I found both, a lean, open-source C preprocessor library, and a reusable YACC grammar for ANSI-C. That was the beginning of the development of Convert::Binary::C in late 2001.

Now, I'm successfully using the module in my embedded environment since long before it appeared on CPAN. From my point of view, it is exactly what I had in mind. It's fast, flexible, easy to use and portable. It doesn't require external programs or other Perl modules.

1.2.2 About this document

This document describes how to use Convert::Binary::C. A lot of different features are presented, and the example code sometimes uses Perl's more advanced language elements. If your experience with Perl is rather limited, you should know how to use Perl's very good documentation system.

To look up one of the manpages, use the `perldoc` command. For example,

```
perldoc perl
```

will show you Perl's main manpage. To look up a specific Perl function, use `perldoc -f`:

```
perldoc -f map
```

gives you more information about the `map` function. You can also search the FAQ using `perldoc -q`:

```
perldoc -q array
```

will give you everything you ever wanted to know about Perl arrays. But now, let's go on with some real stuff!

1.2.3 Why use Convert::Binary::C?

Say you want to pack (or unpack) data according to the following C structure:

```
struct foo {
  char ary[3];
  unsigned short baz;
  int bar;
};
```

You could of course use Perl's `pack` and `unpack` functions:

```
@ary = (1, 2, 3);
$baz = 40000;
$bar = -4711;
$binary = pack 'c3 S i', @ary, $baz, $bar;
```

But this implies that the struct members are byte aligned. If they were long aligned (which is the default for most compilers), you'd have to write

```
$binary = pack 'c3 x S x2 i', @ary, $baz, $bar;
```

which doesn't really increase readability.

Now imagine that you need to pack the data for a completely different architecture with different byte order. You would look into the `pack` manpage again and perhaps come up with this:

```
$binary = pack 'c3 x n x2 N', @ary, $baz, $bar;
```

However, if you try to unpack `$foo` again, your signed values have turned into unsigned ones.

All this can still be managed with Perl. But imagine your structures get more complex? Imagine you need to support different platforms? Imagine you need to make changes to the structures? You'll not only have to change the C source but also dozens of `pack` strings in your Perl code. This is no fun. And Perl should be fun.

Now, wouldn't it be great if you could just read in the C source you've already written and use all the types defined there for packing and unpacking? That's what Convert::Binary::C does.

1.2.4 Creating a Convert::Binary::C object

To use Convert::Binary::C just say

```
use Convert::Binary::C;
```

to load the module. Its interface is completely object oriented, so it doesn't export any functions.

Next, you need to create a new Convert::Binary::C object. This can be done by either

```
$c = Convert::Binary::C->new;
```

or

```
$c = new Convert::Binary::C;
```

You can optionally pass configuration options to the constructor as described in the next section.

1.2.5 Configuring the object

To configure a Convert::Binary::C object, you can either call the `configure` method or directly pass the configuration options to the constructor. If you want to change byte order and alignment, you can use

```
$c->configure(ByteOrder => 'LittleEndian',
              Alignment => 2);
```

or you can change the construction code to

```
$c = new Convert::Binary::C ByteOrder => 'LittleEndian',
                            Alignment => 2;
```

Either way, the object will now know that it should use little endian (Intel) byte order and 2-byte struct member alignment for packing and unpacking.

Alternatively, you can use the option names as names of methods to configure the object, like:

```
$c->ByteOrder('LittleEndian');
```

You can also retrieve information about the current configuration of a Convert::Binary::C object. For details, see the section about the `configure` method.

1.2.6 Parsing C code

Convert::Binary::C allows two ways of parsing C source. Either by parsing external C header or C source files:

```
$c->parse_file('header.h');
```

Or by parsing C code embedded in your script:

```
$c->parse(<<'CCODE');
struct foo {
  char ary[3];
  unsigned short baz;
  int bar;
};
CCODE
```

Now the object `$c` will know everything about `struct foo`. The example above uses a so-called here-document. It allows to easily embed multi-line strings in your code. You can find more about here-documents in the *perldata* manpage or the *perlop* manpage.

Since the `parse` and `parse_file` methods throw an exception when a parse error occurs, you usually want to catch these in an `eval` block:

```
eval { $c->parse_file('header.h') };
if ($@) {
  # handle error appropriately
}
```

Perl's special `$@` variable will contain an empty string (which evaluates to a false value in boolean context) on success or an error string on failure.

As another feature, `parse` and `parse_file` return a reference to their object on success, just like `configure` does when you're configuring the object. This will allow you to write constructs like this:

```
my $c = eval {
  Convert::Binary::C->new(Include => ['/usr/include'])
                    ->parse_file('header.h')
};
if ($@) {
  # handle error appropriately
}
```

1.2.7 Packing and unpacking

Convert::Binary::C has two methods, `pack` and `unpack`, that act similar to the functions of same denominator in Perl. To perform the packing described in the example above, you could write:

```
$data = {
  ary => [1, 2, 3],
  baz => 40000,
  bar => -4711,
```

```
};
$binary = $c->pack('foo', $data);
```

Unpacking will work exactly the same way, just that the unpack method will take a byte string as its input and will return a reference to a (possibly very complex) Perl data structure.

```
$binary = get_data_from_memory();
$data = $c->unpack('foo', $binary);
```

You can now easily access all of the values:

```
print "foo.ary[1] = $data->{ary}[1]\n";
```

Or you can even more conveniently use the Data::Dumper module:

```
use Data::Dumper;
print Dumper($data);
```

The output would look something like this:

```
$VAR1 = {
  'bar' => -271,
  'baz' => 5000,
  'ary' => [
    42,
    48,
    100
  ]
};
```

1.2.8 Preprocessor configuration

Convert::Binary::C uses Thomas Pornin's `ucpp` as an internal C preprocessor. It is compliant to ISO-C99, so you don't have to worry about using even weird preprocessor constructs in your code.

If your C source contains includes or depends upon preprocessor defines, you may need to configure the internal preprocessor. Use the `Include` and `Define` configuration options for that:

```
$c->configure(Include => ['/usr/include',
                          '/home/mhx/include'],
              Define  => [qw( NDEBUG FOO=42 )]);
```

If your code uses system includes, it is most likely that you will need to define the symbols that are usually defined by the compiler.

On some operating systems, the system includes require the preprocessor to predefine a certain set of assertions. Assertions are supported by `ucpp`, and you can define them either in the source code using `#assert` or as a property of the Convert::Binary::C object using `Assert`:

```
$c->configure(Assert => ['predicate(answer)']);
```

Information about defined macros can be retrieved from the preprocessor as long as its configuration isn't changed. The preprocessor is implicitly reset if you change one of the following configuration options:

```
Include
Define
Assert
HasCPPComments
HasMacroVAARGS
```

1.2.9 Supported pragma directives

Convert::Binary::C supports the `pack` pragma to locally override struct member alignment. The supported syntax is as follows:

#pragma pack(ALIGN)

Sets the new alignment to ALIGN. If ALIGN is 0, resets the alignment to its original value.

#pragma pack

Resets the alignment to its original value.

#pragma pack(push, ALIGN)

Saves the current alignment on a stack and sets the new alignment to ALIGN. If ALIGN is 0, sets the alignment to the default alignment.

#pragma pack(pop)

Restores the alignment to the last value saved on the stack.

```
/*  Example assumes sizeof( short ) == 2, sizeof( long ) == 4.  */

#pragma pack(1)

struct nopad {
  char a;               /* no padding bytes between 'a' and 'b' */
  long b;
};

#pragma pack            /* reset to "native" alignment          */

#pragma pack( push, 2 )

struct pad {
  char    a;            /* one padding byte between 'a' and 'b' */
  long    b;

#pragma pack( push, 1 )

  struct {
    char  c;            /* no padding between 'c' and 'd'       */
    short d;
  }       e;            /* sizeof( e ) == 3                     */
```

```
#pragma pack( pop );    /* back to pack( 2 )                    */

  long    f;            /* one padding byte between 'e' and 'f' */
};

#pragma pack( pop );    /* back to "native"                     */
```

The `pack` pragma as it is currently implemented only affects the *maximum* struct member alignment. There are compilers that also allow to specify the *minimum* struct member alignment. This is not supported by Convert::Binary::C.

1.2.10 Automatic configuration using `ccconfig`

As there are over 20 different configuration options, setting all of them correctly can be a lengthy and tedious task.

The `ccconfig` script, which is bundled with this module, aims at automatically determining the correct compiler configuration by testing the compiler executable. It works for both, native and cross compilers.

1.3 Understanding Types

This section covers one of the fundamental features of Convert::Binary::C. It's how *type expressions*, referred to as TYPEs in the method reference, are handled by the module.

Many of the methods, namely `pack`, `unpack`, `sizeof`, `typeof`, `member`, `offsetof`, `def`, `initializer` and `tag`, are passed a TYPE to operate on as their first argument.

1.3.1 Standard Types

These are trivial. Standard types are simply enum names, struct names, union names, or typedefs. Almost every method that wants a TYPE will accept a standard type.

For enums, structs and unions, the prefixes `enum`, `struct` and `union` are optional. However, if a typedef with the same name exists, like in

```
struct foo {
  int bar;
};

typedef int foo;
```

you will have to use the prefix to distinguish between the struct and the typedef. Otherwise, a typedef is always given preference.

1.3.2 Basic Types

Basic types, or atomic types, are `int` or `char`, for example. It's possible to use these basic types without having parsed any code. You can simply do

```
$c = new Convert::Binary::C;
$size = $c->sizeof('unsigned long');
$data = $c->pack('short int', 42);
```

Even though the above works fine, it is not possible to define more complex types on the fly, so

```
$size = $c->sizeof('struct { int a, b; }');
```

will result in an error.

Basic types are not supported by all methods. For example, it makes no sense to use `member` or `offsetof` on a basic type. Using `typeof` isn't very useful, but supported.

1.3.3 Member Expressions

This is by far the most complex part, depending on the complexity of your data structures. Any standard type that defines a compound or an array may be followed by a member expression to select only a certain part of the data type. Say you have parsed the following C code:

```
struct foo {
  long type;
  struct {
    short x, y;
  } array[20];
};

typedef struct foo matrix[8][8];
```

You may want to know the size of the `array` member of `struct foo`. This is quite easy:

```
print $c->sizeof('foo.array'), " bytes";
```

will print

```
80 bytes
```

depending of course on the `ShortSize` you configured.

If you wanted to unpack only a single column of `matrix`, that's easy as well (and of course it doesn't matter which index you use):

```
$column = $c->unpack('matrix[2]', $data);
```

Just like in C, it is possible to use out-of-bounds array indices. This means that, for example, despite `array` is declared to have 20 elements, the following code

```
$size   = $c->sizeof('foo.array[4711]');
$offset = $c->offsetof('foo', 'array[-13]');
```

is perfectly valid and will result in:

```
$size   = 4
$offset = -48
```

Member expressions can be arbitrarily complex:

```
$type = $c->typeof('matrix[2][3].array[7].y');
print "the type is $type";
```

will, for example, print

```
the type is short
```

Member expressions are also used as the second argument to `offsetof`.

1.3.4 Offsets

Members returned by the `member` method have an optional offset suffix to indicate that the given offset doesn't point to the start of that member. For example,

```
$member = $c->member('matrix', 1431);
print $member;
```

will print

```
[2][1].type+3
```

If you would use this as a member expression, like in

```
$size = $c->sizeof("matrix $member");
```

the offset suffix will simply be ignored. Actually, it will be ignored for all methods if it's used in the first argument.

When used in the second argument to `offsetof`, it will usually do what you mean, i. e. the offset suffix, if present, will be considered when determining the offset. This behaviour ensures that

```
$member = $c->member('foo', 43);
$offset = $c->offsetof('foo', $member);
print "'$member' is located at offset $offset of struct foo";
```

will always correctly set `$offset`:

```
'.array[9].y+1' is located at offset 43 of struct foo
```

If this is not what you mean, e.g. because you want to know the offset where the member returned by `member` starts, you just have to remove the suffix:

```
$member =~ s/\+\d+$//;
$offset = $c->offsetof('foo', $member);
print "'$member' starts at offset $offset of struct foo";
```

This would then print:

```
'.array[9].y' starts at offset 42 of struct foo
```

1.4 Using Tags

In a nutshell, tags are properties that you can attach to types.

You can add tags to types using the tag method, and remove them using tag or untag, for example:

```
# Attach 'Format' and 'Hooks' tags
$c->tag('type', Format => 'String', Hooks => { pack => \&rout });

$c->untag('type', 'Format');  # Remove only 'Format' tag
$c->untag('type');            # Remove all tags
```

You can also use tag to see which tags are attached to a type, for example:

```
$tags = $c->tag('type');
```

This would give you:

```
$tags = {
  'Hooks' => {
    'pack' => \&rout
  },
  'Format' => 'String'
};
```

Currently, there are only a couple of different tags that influence the way data is packed and unpacked. There are probably more tags to come in the future.

1.4.1 The Format Tag

One of the tags currently available is the `Format` tag. Using this tag, you can tell a Convert::Binary::C object to pack and unpack a certain data type in a special way.

For example, if you have a (fixed length) string type

```
typedef char str_type[40];
```

this type would, by default, be unpacked as an array of `char`s. That's because it **is** only an array of `char`s, and Convert::Binary::C doesn't know it is actually used as a string.

But you can tell Convert::Binary::C that `str_type` is a C string using the `Format` tag:

```
$c->tag('str_type', Format => 'String');
```

This will make unpack (and of course also pack) treat the binary data like a null-terminated C string:

```
$binary = "Hello World!\n\0 this is just some dummy data";
$hello = $c->unpack('str_type', $binary);
print $hello;
```

would thusly print:

```
Hello World!
```

Of course, this also works the other way round:

```
use Data::Hexdumper;

$binary = $c->pack('str_type', "Just another C::B::C hacker");
print hexdump(data => $binary);
```

would print:

```
  0x0000 : 4A 75 73 74 20 61 6E 6F 74 68 65 72 20 43 3A 3A : Just.another.C::
  0x0010 : 42 3A 3A 43 20 68 61 63 6B 65 72 00 00 00 00 00 : B::C.hacker.....
  0x0020 : 00 00 00 00 00 00 00 00                         : ........
```

If you want Convert::Binary::C to not interpret the binary data at all, you can set the `Format` tag to `Binary`. This might not be seem very useful, as pack and unpack would just pass through the unmodified binary data. But you can tag not only whole types, but also compound members. For example

```
$c->parse(<<ENDC);
struct packet {
  unsigned short header;
  unsigned short flags;
  unsigned char  payload[28];
};
ENDC

$c->tag('packet.payload', Format => 'Binary');
```

would allow you to write:

```
read FILE, $payload, $c->sizeof('packet.payload');

$packet = {
            header  => 4711,
            flags   => 0xf00f,
            payload => $payload,
          };

$binary = $c->pack('packet', $packet);

print hexdump(data => $binary);
```

This would print something like:

```
  0x0000 : 12 67 F0 0F 6E 6F 0A 6E 6F 0A 6E 6F 0A 6E 6F 0A : .g..no.no.no.no.
  0x0010 : 6E 6F 0A 6E 6F 0A 6E 6F 0A 6E 6F 0A 6E 6F 0A 6E : no.no.no.no.no.n
```

For obvious reasons, it is not allowed to attach a `Format` tag to bitfield members. Trying to do so will result in an exception being thrown by the tag method.

1.4.2 The ByteOrder Tag

The `ByteOrder` tag allows you to override the byte order of certain types or members. The implementation of this tag is considered **experimental** and may be subject to changes in the future.

Usually it doesn't make much sense to override the byte order, but there may be applications where a sub-structure is packed in a different byte order than the surrounding structure.

Take, for example, the following code:

```
$c = Convert::Binary::C->new(ByteOrder => 'BigEndian',
                             OrderMembers => 1);
$c->parse(<<'ENDC');

typedef unsigned short u_16;

struct coords_3d {
  long x, y, z;
};

struct coords_msg {
  u_16 header;
  u_16 length;
  struct coords_3d coords;
};

ENDC
```

Assume that while `coords_msg` is big endian, the embedded coordinates `coords_3d` are stored in little endian format for some reason. In C, you'll have to handle this manually.

But using Convert::Binary::C, you can simply attach a `ByteOrder` tag to either the `coords_3d` structure or to the `coords` member of the `coords_msg` structure. Both will work in this case. The only difference is that if you tag the `coords` member, `coords_3d` will only be treated as little endian if you `pack` or `unpack` the `coords_msg` structure. (BTW, you could also tag all members of `coords_3d` individually, but that would be inefficient.)

So, let's attach the `ByteOrder` tag to the `coords` member:

```
$c->tag('coords_msg.coords', ByteOrder => 'LittleEndian');
```

Assume the following binary message:

```
0x0000 : 00 2A 00 0C FF FF FF FF 02 00 00 00 2A 00 00 00 : .*..........*...
```

If you unpack this message...

```
$msg = $c->unpack('coords_msg', $binary);
```

...you will get the following data structure:

```
$msg = {
  'header' => 42,
  'length' => 12,
  'coords' => {
    'x' => -1,
    'y' => 2,
    'z' => 42
  }
};
```

Without the `ByteOrder` tag, you would get:

```
$msg = {
  'header' => 42,
  'length' => 12,
  'coords' => {
    'x' => -1,
    'y' => 33554432,
    'z' => 704643072
  }
};
```

The `ByteOrder` tag is a *recursive* tag, i.e. it applies to all children of the tagged object recursively. Of course, it is also possible to override a `ByteOrder` tag by attaching another `ByteOrder` tag to a child type. Confused? Here's an example. In addition to tagging the `coords` member as little endian, we now tag `coords_3d.y` as big endian:

```
$c->tag('coords_3d.y', ByteOrder => 'BigEndian');
$msg = $c->unpack('coords_msg', $binary);
```

This will return the following data structure:

```
$msg = {
  'header' => 42,
  'length' => 12,
  'coords' => {
    'x' => -1,
    'y' => 33554432,
    'z' => 42
  }
};
```

Note that if you tag both a type and a member of that type within a compound, the tag attached to the type itself has higher precedence. Using the example above, if you would attach a `ByteOrder` tag to both `coords_msg.coords` and `coords_3d`, the tag attached to `coords_3d` would always win.

Also note that the `ByteOrder` tag might not work as expected along with bitfields, which is why the implementation is considered experimental. Bitfields are currently **not** affected by the `ByteOrder` tag at all. This is because the byte order would affect the bitfield layout, and a consistent implementation supporting multiple layouts of the same struct would be quite bulky and probably slow down the whole module.

If you really need the correct behaviour, you can use the following trick:

```
$le = Convert::Binary::C->new(ByteOrder => 'LittleEndian');

$le->parse(<<'ENDC');

typedef unsigned short u_16;
typedef unsigned long  u_32;

struct message {
  u_16 header;
  u_16 length;
  struct {
    u_32 a;
    u_32 b;
    u_32 c :  7;
    u_32 d :  5;
    u_32 e : 20;
  } data;
};

ENDC

$be = $le->clone->ByteOrder('BigEndian');

$le->tag('message.data', Format => 'Binary', Hooks => {
    unpack => sub { $be->unpack('message.data', @_) },
    pack   => sub { $be->pack('message.data', @_) },
  });

$msg = $le->unpack('message', $binary);
```

This uses the `Format` and `Hooks` tags along with a big endian `clone` of the original little endian object. It attaches hooks to the little endian object and in the hooks it uses the big endian object to `pack` and `unpack` the binary data.

1.4.3 The Dimension Tag

The `Dimension` tag allows you to override the declared dimension of an array for packing or unpacking data. The implementation of this tag is considered **very experimental** and will **definitely change** in a future release.

That being said, the `Dimension` tag is primarily useful to support variable length arrays. Usually, you have to write the following code for such a variable length array in C:

```
struct c_message
{
  unsigned count;
  char data[1];
};
```

So, because you cannot declare an empty array, you declare an array with a single element. If you have a ISO-C99 compliant compiler, you can write this code instead:

```
struct c99_message
{
  unsigned count;
  char data[];
};
```

This explicitly tells the compiler that `data` is a flexible array member. Convert::Binary::C already uses this information to handle flexible array members in a special way.

As you can see in the following example, the two types are treated differently:

```
$data = pack 'NC*', 3, 1..8;
$uc   = $c->unpack('c_message', $data);
$uc99 = $c->unpack('c99_message', $data);
```

This will result in:

```
$uc = {'count' => 3,'data' => [1]};
$uc99 = {'count' => 3,'data' => [1,2,3,4,5,6,7,8]};
```

However, only few compilers support ISO-C99, and you probably don't want to change your existing code only to get some extra features when using Convert::Binary::C.

So it is possible to attach a tag to the `data` member of the `c_message` struct that tells Convert::Binary::C to treat the array as if it were flexible:

```
$c->tag('c_message.data', Dimension => '*');
```

Now both `c_message` and `c99_message` will behave exactly the same when using `pack` or `unpack`. Repeating the above code:

```
$uc = $c->unpack('c_message', $data);
```

This will result in:

```
$uc = {'count' => 3,'data' => [1,2,3,4,5,6,7,8]};
```

But there's more you can do. Even though it probably doesn't make much sense, you can tag a fixed dimension to an array:

```
$c->tag('c_message.data', Dimension => '5');
```

This will obviously result in:

```
$uc = {'count' => 3,'data' => [1,2,3,4,5]};
```

A more useful way to use the `Dimension` tag is to set it to the name of a member in the same compound:

```
$c->tag('c_message.data', Dimension => 'count');
```

Convert::Binary::C will now use the value of that member to determine the size of the array, so unpacking will result in:

```
$uc = {'count' => 3,'data' => [1,2,3]};
```

Of course, you can also tag flexible array members. And yes, it's also possible to use more complex member expressions:

```
$c->parse(<<ENDC);
struct msg_header
{
  unsigned len[2];
};

struct more_complex
{
  struct msg_header hdr;
  char data[];
};
ENDC

$data = pack 'NNC*', 42, 7, 1 .. 10;

$c->tag('more_complex.data', Dimension => 'hdr.len[1]');

$u = $c->unpack('more_complex', $data);
```

The result will be:

```
$u = {
  'hdr' => {
    'len' => [
      42,
      7
    ]
  },
  'data' => [
    1,
    2,
    3,
    4,
    5,
    6,
    7
  ]
};
```

By the way, it's also possible to tag arrays that are not embedded inside a compound:

```
$c->parse(<<ENDC);
typedef unsigned short short_array[];
ENDC

$c->tag('short_array', Dimension => '5');
```

```
$u = $c->unpack('short_array', $data);
```

Resulting in:

```
$u = [0,42,0,7,258];
```

The final and most powerful way to define a `Dimension` tag is to pass it a subroutine reference. The referenced subroutine can execute whatever code is neccessary to determine the size of the tagged array:

```
sub get_size
{
  my $m = shift;
  return $m->{hdr}{len}[0] / $m->{hdr}{len}[1];
}

$c->tag('more_complex.data', Dimension => \&get_size);

$u = $c->unpack('more_complex', $data);
```

As you can guess from the above code, the subroutine is being passed a reference to hash that stores the already unpacked part of the compound embedding the tagged array. This is the result:

```
$u = {
  'hdr' => {
    'len' => [
      42,
      7
    ]
  },
  'data' => [
    1,
    2,
    3,
    4,
    5,
    6
  ]
};
```

You can also pass custom arguments to the subroutines by using the `arg` method. This is similar to the functionality offered by the `Hooks` tag.

Of course, all that also works for the `pack` method as well.

However, the current implementation has at least one shortcomings, which is why it's experimental: The `Dimension` tag doesn't impact compound layout. This means that while you can alter the size of an array in the middle of a compound, the offset of the members after that array won't be impacted. I'd rather like to see the layout adapt dynamically, so this is what I'm hoping to implement in the future.

1.4.4 The Hooks Tag

Hooks are a special kind of tag that can be extremely useful.

Using hooks, you can easily override the way pack and unpack handle data using your own subroutines. If you define hooks for a certain data type, each time this data type is processed the corresponding hook will be called to allow you to modify that data.

Basic Hooks

Here's an example. Let's assume the following C code has been parsed:

```
typedef unsigned long u_32;
typedef u_32          ProtoId;
typedef ProtoId       MyProtoId;

struct MsgHeader {
  MyProtoId id;
  u_32      len;
};

struct String {
  u_32 len;
  char buf[];
};
```

You could now use the types above and, for example, unpack binary data representing a `MsgHeader` like this:

```
$msg_header = $c->unpack('MsgHeader', $data);
```

This would give you:

```
$msg_header = {
  'len' => 13,
  'id' => 42
};
```

Instead of dealing with `ProtoId`'s as integers, you would rather like to have them as clear text. You could provide subroutines to convert between clear text and integers:

```
%proto = (
  CATS      =>    1,
  DOGS      =>   42,
  HEDGEHOGS => 4711,
);

%rproto = reverse %proto;

sub ProtoId_unpack {
  $rproto{$_[0]} || 'unknown protocol'
}
```

```
sub ProtoId_pack {
  $proto{$_[0]} or die 'unknown protocol'
}
```

You can now register these subroutines by attaching a `Hooks` tag to `ProtoId` using the `tag` method:

```
$c->tag('ProtoId', Hooks => { pack   => \&ProtoId_pack,
                              unpack => \&ProtoId_unpack });
```

Doing exactly the same unpack on `MsgHeader` again would now return:

```
$msg_header = {
  'len' => 13,
  'id' => 'DOGS'
};
```

Actually, if you don't need the reverse operation, you don't even have to register a `pack` hook. Or, even better, you can have a more intelligent `unpack` hook that creates a dual-typed variable:

```
use Scalar::Util qw(dualvar);

sub ProtoId_unpack2 {
  dualvar $_[0], $rproto{$_[0]} || 'unknown protocol'
}

$c->tag('ProtoId', Hooks => { unpack => \&ProtoId_unpack2 });

$msg_header = $c->unpack('MsgHeader', $data);
```

Just as before, this would print

```
$msg_header = {
  'len' => 13,
  'id' => 'DOGS'
};
```

but without requiring a `pack` hook for packing, at least as long as you keep the variable dual-typed.

Hooks are usually called with exactly one argument, which is the data that should be processed (see the section on Advanced Hooks on page 29 for details on how to customize hook arguments). They are called in scalar context and expected to return the processed data.

To get rid of registered hooks, you can either undefine only certain hooks

```
$c->tag('ProtoId', Hooks => { pack => undef });
```

or all hooks:

```
$c->tag('ProtoId', Hooks => undef);
```

Of course, hooks are not restricted to handling integer values. You could just as well attach hooks for the `String` struct from the code above. A useful example would be to have these hooks:

```
sub string_unpack {
  my $s = shift;
  pack "c$s->{len}", @{$s->{buf}};
}

sub string_pack {
  my $s = shift;
  return {
    len => length $s,
    buf => [ unpack 'c*', $s ],
  }
}
```

(Don't be confused by the fact that the `unpack` hook uses `pack` and the `pack` hook uses `unpack`. And also see the section on Advanced Hooks on page 29 for a more clever approach.)

While you would normally get the following output when unpacking a `String`

```
$string = {
  'len' => 12,
  'buf' => [
    72,
    101,
    108,
    108,
    111,
    32,
    87,
    111,
    114,
    108,
    100,
    33
  ]
};
```

you could just register the hooks using

```
$c->tag('String', Hooks => { pack   => \&string_pack,
                             unpack => \&string_unpack });
```

and you would get a nice human-readable Perl string:

```
$string = 'Hello World!';
```

Packing a string turns out to be just as easy:

```
use Data::Hexdumper;

$data = $c->pack('String', 'Just another Perl hacker,');

print hexdump(data => $data);
```

This would print:

```
0x0000 : 00 00 00 19 4A 75 73 74 20 61 6E 6F 74 68 65 72 : ....Just.another
0x0010 : 20 50 65 72 6C 20 68 61 63 6B 65 72 2C          : .Perl.hacker,
```

If you want to find out if or which hooks are registered for a certain type, you can also use the `tag` method:

```
$hooks = $c->tag('String', 'Hooks');
```

This would return:

```
$hooks = {
  'unpack' => \&string_unpack,
  'pack' => \&string_pack
};
```

Advanced Hooks

It is also possible to combine hooks with using the `Format` tag. This can be useful if you know better than Convert::Binary::C how to interpret the binary data. In the previous section, we've handled this type

```
struct String {
  u_32 len;
  char buf[];
};
```

with the following hooks:

```
sub string_unpack {
  my $s = shift;
  pack "c$s->{len}", @{$s->{buf}};
}

sub string_pack {
  my $s = shift;
  return {
    len => length $s,
    buf => [ unpack 'c*', $s ],
  }
}

$c->tag('String', Hooks => { pack   => \&string_pack,
                             unpack => \&string_unpack });
```

As you can see in the hook code, `buf` is expected to be an array of characters. For the `unpack` case Convert::Binary::C first turns the binary data into a Perl array, and then the hook packs it back into a string. The intermediate array creation and destruction is completely useless. Same thing, of course, for the `pack` case.

Here's a clever way to handle this. Just tag `buf` as binary

```
$c->tag('String.buf', Format => 'Binary');
```

and use the following hooks instead:

```
sub string_unpack2 {
  my $s = shift;
  substr $s->{buf}, 0, $s->{len};
}

sub string_pack2 {
  my $s = shift;
  return {
    len => length $s,
    buf => $s,
  }
}

$c->tag('String', Hooks => { pack   => \&string_pack2,
                             unpack => \&string_unpack2 });
```

This will be exactly equivalent to the old code, but faster and probably even much easier to understand.

But hooks are even more powerful. You can customize the arguments that are passed to your hooks and you can use arg to pass certain special arguments, such as the name of the type that is currently being processed by the hook.

The following example shows how it is easily possible to peek into the perl internals using hooks.

```
use Config;

$c = new Convert::Binary::C %CC, OrderMembers => 1;
$c->Include(["$Config{archlib}/CORE", @{$c->Include}]);
$c->parse(<<ENDC);
#include "EXTERN.h"
#include "perl.h"
ENDC

$c->tag($_, Hooks => { unpack_ptr => [\&unpack_ptr,
                                      $c->arg(qw(SELF TYPE DATA))] })
    for qw( XPVAV XPVHV MAGIC MGVTBL HV );
```

First, we add the perl core include path and parse *perl.h*. Then, we add an `unpack_ptr` hook for a couple of the internal data types.

The `unpack_ptr` and `pack_ptr` hooks are called whenever a pointer to a certain data structure is processed. This is by far the most experimental part of the hooks feature, as this includes **any** kind of pointer. There's no way for the hook to know the difference between a plain pointer, or a pointer to a pointer, or a pointer to an array (this is because the difference doesn't matter anywhere else in Convert::Binary::C).

But the hook above makes use of another very interesting feature: It uses arg to pass special arguments to the hook subroutine. Usually, the hook subroutine is simply passed a single data argument. But using the above definition, it'll get a reference to the calling object (`SELF`), the name of the type being processed (`TYPE`) and the data (`DATA`).

But how does our hook look like?

```
sub unpack_ptr {
  my($self, $type, $ptr) = @_;
  $ptr or return '<NULL>';
  my $size = $self->sizeof($type);
  $self->unpack($type, unpack("P$size", pack('I', $ptr)));
}
```

As you can see, the hook is rather simple. First, it receives the arguments mentioned above. It performs a quick check if the pointer is `NULL` and shouldn't be processed any further. Next, it determines the size of the type being processed. And finally, it'll just use the `P`*n* unpack template to read from that memory location and recursively call `unpack` to unpack the type. (And yes, this may of course again call other hooks.)

Now, let's test that:

```
my $ref = bless ["Boo!"], "Foo::Bar";
my $ptr = hex(("$ref" =~ /\(0x([[:xdigit:]]+)\)$/)[0]);

print Dumper(unpack_ptr($c, 'AV', $ptr));
```

Just for the fun of it, we create a blessed array reference. But how do we get a pointer to the corresponding `AV`? This is rather easy, as the address of the `AV` is just the hex value that appears when using the array reference in string context. So we just grab that and turn it into decimal. All that's left to do is just call our hook, as it can already handle `AV` pointers. And this is what we get:

```
$VAR1 = {
  'sv_any' => {
    'xnv_u' => {
      'xnv_nv' => '0',
      'xgv_stash' => '<NULL>',
      'xpad_cop_seq' => {
        'xlow' => 0,
        'xhigh' => 0
      },
      'xbm_s' => {
        'xbm_previous' => 0,
        'xbm_flags' => 0,
        'xbm_rare' => 0
      }
    },
    'xav_fill' => 0,
    'xav_max' => 0,
    'xiv_u' => {
      'xivu_iv' => 140654788,
      'xivu_uv' => 140654788,
      'xivu_p1' => 140654788,
      'xivu_i32' => 140654788,
      'xivu_namehek' => 140654788
    },
    'xmg_u' => {
      'xmg_magic' => '<NULL>',
      'xmg_ourstash' => '<NULL>'
    },
    'xmg_stash' => {
```

```
'sv_any' => {
  'xnv_u' => {
    'xnv_nv' => '0',
    'xgv_stash' => '<NULL>',
    'xpad_cop_seq' => {
      'xlow' => 0,
      'xhigh' => 0
    },
    'xbm_s' => {
      'xbm_previous' => 0,
      'xbm_flags' => 0,
      'xbm_rare' => 0
    }
  },
  'xhv_fill' => 2,
  'xhv_max' => 7,
  'xiv_u' => {
    'xivu_iv' => 2,
    'xivu_uv' => 2,
    'xivu_p1' => 2,
    'xivu_i32' => 2,
    'xivu_namehek' => 2
  },
  'xmg_u' => {
    'xmg_magic' => {
      'mg_moremagic' => '<NULL>',
      'mg_virtual' => {
        'svt_get' => 0,
        'svt_set' => 0,
        'svt_len' => 0,
        'svt_clear' => 0,
        'svt_free' => 136797852,
        'svt_copy' => 0,
        'svt_dup' => 0,
        'svt_local' => 0
      },
      'mg_private' => 0,
      'mg_type' => 99,
      'mg_flags' => 0,
      'mg_len' => 12,
      'mg_obj' => 0,
      'mg_ptr' => 139732148
    },
    'xmg_ourstash' => {
      'sv_any' => '<NULL>',
      'sv_refcnt' => 137499872,
      'sv_flags' => 6488064,
      'sv_u' => {
        'svu_iv' => 12,
        'svu_uv' => 12,
        'svu_rv' => 12,
        'svu_pv' => 12,
```

```
              'svu_array' => 12,
              'svu_hash' => 12,
              'svu_gp' => 12
            }
          }
        },
        'xmg_stash' => '<NULL>'
      },
      'sv_refcnt' => 2,
      'sv_flags' => 578813964,
      'sv_u' => {
        'svu_iv' => '8243107277089939388',
        'svu_uv' => '8243107277089939388',
        'svu_rv' => 140648380,
        'svu_pv' => 140648380,
        'svu_array' => 140648380,
        'svu_hash' => 140648380,
        'svu_gp' => 140648380
      }
    }
  },
  'sv_refcnt' => 1,
  'sv_flags' => 1074790411,
  'sv_u' => {
    'svu_iv' => '592594608112941252',
    'svu_uv' => '592594608112941252',
    'svu_rv' => 140654788,
    'svu_pv' => 140654788,
    'svu_array' => 140654788,
    'svu_hash' => 140654788,
    'svu_gp' => 140654788
  }
};
```

Even though it is rather easy to do such stuff using `unpack_ptr` hooks, you should really know what you're doing and do it with extreme care because of the limitations mentioned above. It's really easy to run into segmentation faults when you're dereferencing pointers that point to memory which you don't own.

Performance

Using hooks isn't for free. In performance-critical applications you have to keep in mind that hooks are actually perl subroutines and that they are called once for every value of a registered type that is being packed or unpacked. If only about 10% of the values require hooks to be called, you'll hardly notice the difference (if your hooks are implemented efficiently, that is). But if all values would require hooks to be called, that alone could easily make packing and unpacking very slow.

1.4.5 Tag Order

Since it is possible to attach multiple tags to a single type, the order in which the tags are processed is important. Here's a small table that shows the processing order.

```
pack        unpack
--------------------
Hooks       Format
Format      ByteOrder
ByteOrder   Hooks
```

As a general rule, the Hooks tag is always the first thing processed when packing data, and the last thing processed when unpacking data.

The Format and ByteOrder tags are exclusive, but when both are given the Format tag wins.

1.5 Methods

1.5.1 new

new

new **OPTION1 => VALUE1, OPTION2 => VALUE2, ...**

The constructor is used to create a new Convert::Binary::C object. You can simply use

```
$c = new Convert::Binary::C;
```

without additional arguments to create an object, or you can optionally pass any arguments to the constructor that are described for the configure method.

1.5.2 configure

configure

configure **OPTION**

configure **OPTION1 => VALUE1, OPTION2 => VALUE2, ...**

This method can be used to configure an existing Convert::Binary::C object or to retrieve its current configuration.

To configure the object, the list of options consists of key and value pairs and must therefore contain an even number of elements. configure (and also new if used with configuration options) will throw an exception if you pass an odd number of elements. Configuration will normally look like this:

```
$c->configure(ByteOrder => 'BigEndian', IntSize => 2);
```

To retrieve the current value of a configuration option, you must pass a single argument to configure that holds the name of the option, just like

```
$order = $c->configure('ByteOrder');
```

If you want to get the values of all configuration options at once, you can call configure without any arguments and it will return a reference to a hash table that holds the whole object configuration. This can be conveniently used with the Data::Dumper module, for example:

```
use Convert::Binary::C;
use Data::Dumper;
```

```
$c = new Convert::Binary::C Define  => ['DEBUGGING', 'FOO=123'],
                            Include => ['/usr/include'];

print Dumper($c->configure);
```

Which will print something like this:

```
$VAR1 = {
  'Define' => [
    'DEBUGGING',
    'FOO=123'
  ],
  'ByteOrder' => 'LittleEndian',
  'LongSize' => 4,
  'IntSize' => 4,
  'ShortSize' => 2,
  'HasMacroVAARGS' => 1,
  'Assert' => [],
  'UnsignedChars' => 0,
  'DoubleSize' => 8,
  'CharSize' => 1,
  'EnumType' => 'Integer',
  'PointerSize' => 4,
  'EnumSize' => 4,
  'DisabledKeywords' => [],
  'FloatSize' => 4,
  'Alignment' => 1,
  'LongLongSize' => 8,
  'LongDoubleSize' => 12,
  'KeywordMap' => {},
  'Include' => [
    '/usr/include'
  ],
  'HasCPPComments' => 1,
  'Bitfields' => {
    'Engine' => 'Generic'
  },
  'UnsignedBitfields' => 0,
  'Warnings' => 0,
  'CompoundAlignment' => 1,
  'OrderMembers' => 0
};
```

Since you may not always want to write a `configure` call when you only want to change a single configuration item, you can use any configuration option name as a method name, like:

```
$c->ByteOrder('LittleEndian') if $c->IntSize < 4;
```

(Yes, the example doesn't make very much sense... ;-)

However, you should keep in mind that configuration methods that can take lists (namely `Include`, `Define` and `Assert`, but not `DisabledKeywords`) may behave slightly different than their `configure` equivalent. If you pass these methods a single argument that is an array reference, the current list will be **replaced** by the new one, which is just the behaviour of the corresponding `configure` call. So the following are equivalent:

```
$c->configure(Define => ['foo', 'bar=123']);
$c->Define(['foo', 'bar=123']);
```

But if you pass a list of strings instead of an array reference (which cannot be done when using configure), the new list items are **appended** to the current list, so

```
$c = new Convert::Binary::C Include => ['/include'];
$c->Include('/usr/include', '/usr/local/include');
print Dumper($c->Include);

$c->Include(['/usr/local/include']);
print Dumper($c->Include);
```

will first print all three include paths, but finally only `/usr/local/include` will be configured:

```
$VAR1 = [
  '/include',
  '/usr/include',
  '/usr/local/include'
];
$VAR1 = [
  '/usr/local/include'
];
```

Furthermore, configuration methods can be chained together, as they return a reference to their object if called as a set method. So, if you like, you can configure your object like this:

```
$c = Convert::Binary::C->new(IntSize => 4)
       ->Define(qw( __DEBUG__ DB_LEVEL=3 ))
       ->ByteOrder('BigEndian');

$c->configure(EnumType => 'Both', Alignment => 4)
  ->Include('/usr/include', '/usr/local/include');
```

In the example above, `qw( ... )` is the word list quoting operator. It returns a list of all non-whitespace sequences, and is especially useful for configuring preprocessor defines or assertions. The following assignments are equivalent:

```
@array = ('one', 'two', 'three');
@array = qw(one two three);
```

You can configure the following options. Unknown options, as well as invalid values for an option, will cause the object to throw exceptions.

`IntSize` => **0** | **1** | **2** | **4** | **8**

Set the number of bytes that are occupied by an integer. This is in most cases 2 or 4. If you set it to zero, the size of an integer on the host system will be used. This is also the default unless overridden by `CBC_DEFAULT_INT_SIZE` at compile time.

`CharSize` => **0** | **1** | **2** | **4** | **8**

Set the number of bytes that are occupied by a `char`. This rarely needs to be changed, except for some platforms that don't care about bytes, for example DSPs. If you set this to zero, the size of a `char` on the host system will be used. This is also the default unless overridden by `CBC_DEFAULT_CHAR_SIZE` at compile time.

`ShortSize` => **0** | **1** | **2** | **4** | **8**

Set the number of bytes that are occupied by a short integer. Although integers explicitly declared as `short` should be always 16 bit, there are compilers that make a short 8 bit wide. If you set it to zero, the size of a short integer on the host system will be used. This is also the default unless overridden by `CBC_DEFAULT_SHORT_SIZE` at compile time.

`LongSize` => **0** | **1** | **2** | **4** | **8**

Set the number of bytes that are occupied by a long integer. If set to zero, the size of a long integer on the host system will be used. This is also the default unless overridden by `CBC_DEFAULT_LONG_SIZE` at compile time.

`LongLongSize` => **0** | **1** | **2** | **4** | **8**

Set the number of bytes that are occupied by a long long integer. If set to zero, the size of a long long integer on the host system, or 8, will be used. This is also the default unless overridden by `CBC_DEFAULT_LONG_LONG_SIZE` at compile time.

`FloatSize` => **0** | **1** | **2** | **4** | **8** | **12** | **16**

Set the number of bytes that are occupied by a single precision floating point value. If you set it to zero, the size of a `float` on the host system will be used. This is also the default unless overridden by `CBC_DEFAULT_FLOAT_SIZE` at compile time. For details on floating point support, see the section on Floating Point Values on page 84.

`DoubleSize` => **0** | **1** | **2** | **4** | **8** | **12** | **16**

Set the number of bytes that are occupied by a double precision floating point value. If you set it to zero, the size of a `double` on the host system will be used. This is also the default unless overridden by `CBC_DEFAULT_DOUBLE_SIZE` at compile time. For details on floating point support, see the section on Floating Point Values on page 84.

`LongDoubleSize` => **0** | **1** | **2** | **4** | **8** | **12** | **16**

Set the number of bytes that are occupied by a double precision floating point value. If you set it to zero, the size of a `long double` on the host system, or 12 will be used. This is also the default unless overridden by `CBC_DEFAULT_LONG_DOUBLE_SIZE` at compile time. For details on floating point support, see the section on Floating Point Values on page 84.

`PointerSize` => **0** | **1** | **2** | **4** | **8**

Set the number of bytes that are occupied by a pointer. This is in most cases 2 or 4. If you set it to zero, the size of a pointer on the host system will be used. This is also the default unless overridden by `CBC_DEFAULT_PTR_SIZE` at compile time.

`EnumSize` => **-1** | **0** | **1** | **2** | **4** | **8**

Set the number of bytes that are occupied by an enumeration type. On most systems, this is equal to the size of an integer, which is also the default. However, for some compilers, the size of an enumeration type depends on the size occupied by the largest enumerator. So the size may vary between 1 and 8. If you have

```
enum foo {
  ONE = 100, TWO = 200
};
```

this will occupy one byte because the enum can be represented as an unsigned one-byte value. However,

```
enum foo {
  ONE = -100, TWO = 200
};
```

will occupy two bytes, because the -100 forces the type to be signed, and 200 doesn't fit into a signed one-byte value. Therefore, the type used is a signed two-byte value. If this is the behaviour you need, set the EnumSize to `0`.

Some compilers try to follow this strategy, but don't care whether the enumeration has signed values or not. They always declare an enum as signed. On such a compiler, given

```
enum one { ONE = -100, TWO = 100 };
enum two { ONE =  100, TWO = 200 };
```

enum `one` will occupy only one byte, while enum `two` will occupy two bytes, even though it could be represented by a unsigned one-byte value. If this is the behaviour of your compiler, set EnumSize to `-1`.

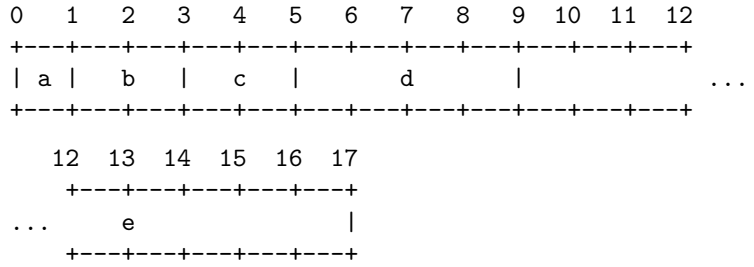
`Alignment` => **0** | **1** | **2** | **4** | **8** | **16**

Set the struct member alignment. This option controls where padding bytes are inserted between struct members. It globally sets the alignment for all structs/unions. However, this can be overridden from within the source code with the common `pack` pragma as explained in the section on Supported pragma directives on page 14. The default alignment is 1, which means no padding bytes are inserted. A setting of `0` means *native* alignment, i.e. the alignment of the system that Convert::Binary::C has been compiled on. You can determine the native properties using the `native` function.

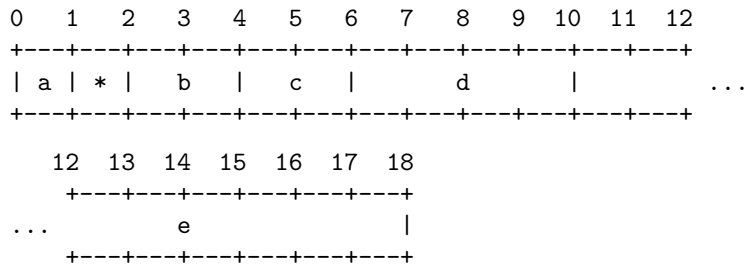
The `Alignment` option is similar to the `-Zp[n]` option of the Intel compiler. It globally specifies the maximum boundary to which struct members are aligned. Consider the following structure and the sizes of `char`, `short`, `long` and `double` being 1, 2, 4 and 8, respectively.

```
struct align {
  char   a;
  short  b, c;
  long   d;
  double e;
};
```

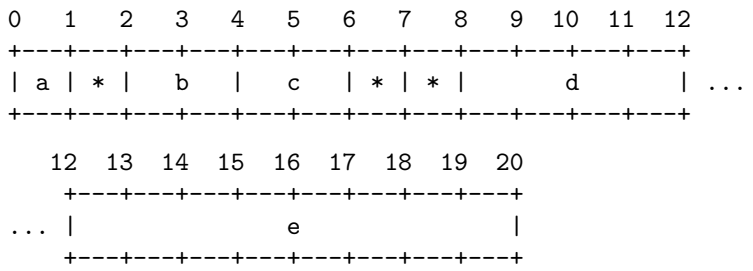
With an alignment of 1 (the default), the struct members would be packed tightly:



With an alignment of 2, the struct members larger than one byte would be aligned to 2-byte boundaries, which results in a single padding byte between `a` and `b`.

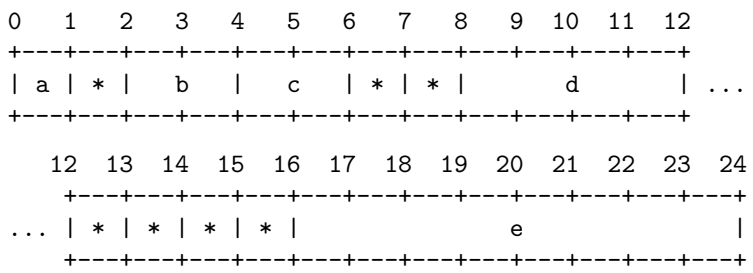


With an alignment of 4, the struct members of size 2 would be aligned to 2-byte boundaries and larger struct members would be aligned to 4-byte boundaries:



This layout of the struct members allows the compiler to generate optimized code because aligned members can be accessed more easily by the underlying architecture.

Finally, setting the alignment to 8 will align `double`s to 8-byte boundaries:



Further increasing the alignment does not alter the layout of our structure, as only members larger that 8 bytes would be affected.

The alignment of a structure depends on its largest member and on the setting of the `Alignment` option. With `Alignment` set to 2, a structure holding a `long` would be aligned to a 2-byte boundary, while a structure containing only `char`s would have no alignment restrictions. (Unfortunately, that's not the whole story. See the `CompoundAlignment` option for details.)

Here's another example. Assuming 8-byte alignment, the following two structs will both have a size of 16 bytes:

```
struct one {
  char   c;
  double d;
};

struct two {
  double d;
  char   c;
};
```

This is clear for `struct one`, because the member `d` has to be aligned to an 8-byte boundary, and thus 7 padding bytes are inserted after `c`. But for `struct two`, the padding bytes are inserted **at the end** of the structure, which doesn't make much sense immediately. However, it makes perfect sense if you think about an array of `struct two`. Each `double` has to be aligned to an 8-byte boundary, an thus each array element would have to occupy 16 bytes. With that in mind, it would be strange if a `struct two` variable would have a different size. And it would make the widely used construct

```
struct two array[] = { {1.0, 0}, {2.0, 1} };
int elements = sizeof(array) / sizeof(struct two);
```

impossible.

The alignment behaviour described here seems to be common for all compilers. However, not all compilers have an option to configure their default alignment.

`CompoundAlignment` => **0** | **1** | **2** | **4** | **8** | **16**

Usually, the alignment of a compound (i.e. a `struct` or a `union`) depends only on its largest member and on the setting of the `Alignment` option. There are, however, architectures and compilers where compounds can have different alignment constraints.

For most platforms and compilers, the alignment constraint for compounds is 1 byte. That is, on most platforms

```
struct onebyte {
  char byte;
};
```

will have an alignment of 1 and also a size of 1. But if you take an ARM architecture, the above `struct onebyte` will have an alignment of 4, and thus also a size of 4.

You can configure this by setting `CompoundAlignment` to 4. This will ensure that the alignment of compounds is always 4.

Setting `CompoundAlignment` to `0` means *native* compound alignment, i.e. the compound alignment of the system that Convert::Binary::C has been compiled on. You can determine the native properties using the `native` function.

There are also compilers for certain platforms that allow you to adjust the compound alignment. If you're not aware of the fact that your compiler/architecture has a compound alignment other than 1, strange things can happen. If, for example, the compound alignment is 2 and you have something like

```
typedef unsigned char U8;

struct msg_head {
  U8 cmd;
  struct {
    U8 hi;
    U8 low;
  } crc16;
  U8 len;
};
```

there will be one padding byte inserted before the embedded `crc16` struct and after the `len` member, which is most probably not what was intended:

```
0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+
| cmd |  *  | hi  | low | len |  *  |
+-----+-----+-----+-----+-----+-----+
```

Note that both `#pragma pack` and the `Alignment` option can override `CompoundAlignment`. If you set `CompoundAlignment` to 4, but `Alignment` to 2, compounds will actually be aligned on 2-byte boundaries.

`ByteOrder` => **'BigEndian'** | **'LittleEndian'**

Set the byte order for integers larger than a single byte. Little endian (Intel, least significant byte first) and big endian (Motorola, most significant byte first) byte order are supported. The default byte order is the same as the byte order of the host system unless overridden by `CBC_DEFAULT_BYTEORDER` at compile time.

`EnumType` => **'Integer'** | **'String'** | **'Both'**

This option controls the type that enumeration constants will have in data structures returned by the `unpack` method. If you have the following definitions:

```
typedef enum {
  SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
  THURSDAY, FRIDAY, SATURDAY
} Weekday;

typedef enum {
  JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY,
  AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER
} Month;

typedef struct {
  int     year;
  Month   month;
  int     day;
  Weekday weekday;
} Date;
```

and a byte string that holds a packed Date struct, then you'll get the following results from a call to the unpack method.

`Integer`
: Enumeration constants are returned as plain integers. This is fast, but may be not very useful. It is also the default.

```
$date = {
  'weekday' => 1,
  'month' => 0,
  'day' => 7,
  'year' => 2002
};
```

`String`
: Enumeration constants are returned as strings. This will create a string constant for every unpacked enumeration constant and thus consumes more time and memory. However, the result may be more useful.

```
$date = {
  'weekday' => 'MONDAY',
  'month' => 'JANUARY',
  'day' => 7,
  'year' => 2002
};
```

`Both`
: Enumeration constants are returned as double typed scalars. If evaluated in string context, the enumeration constant will be a string, if evaluated in numeric context, the enumeration constant will be an integer.

```
$date = $c->EnumType('Both')->unpack('Date', $binary);

printf "Weekday = %s (%d)\n\n", $date->{weekday},
                                $date->{weekday};

if ($date->{month} == 0) {
  print "It's $date->{month}, happy new year!\n\n";
}

print Dumper($date);
```

This will print:

```
Weekday = MONDAY (1)
```

```
It's JANUARY, happy new year!
$VAR1 = {
  'weekday' => 'MONDAY',
  'month' => 'JANUARY',
  'day' => 7,
  'year' => 2002
};
```

`DisabledKeywords` => [**KEYWORDS**]

This option allows you to selectively deactivate certain keywords in the C parser. Some C compilers don't have the complete ANSI keyword set, i.e. they don't recognize the keywords `const` or `void`, for example. If you do

```
typedef int void;
```

on such a compiler, this will usually be ok. But if you parse this with an ANSI compiler, it will be a syntax error. To parse the above code correctly, you have to disable the `void` keyword in the Convert::Binary::C parser:

```
$c->DisabledKeywords([qw( void )]);
```

By default, the Convert::Binary::C parser will recognize the keywords `inline` and `restrict`. If your compiler doesn't have these new keywords, it usually doesn't matter. Only if you're using the keywords as identifiers, like in

```
typedef struct inline {
  int a, b;
} restrict;
```

you'll have to disable these ISO-C99 keywords:

```
$c->DisabledKeywords([qw( inline restrict )]);
```

The parser allows you to disable the following keywords:

```
asm
auto
const
double
enum
extern
float
inline
long
register
restrict
short
signed
static
unsigned
void
volatile
```

`KeywordMap` => { **KEYWORD** => **TOKEN, ...** }

This option allows you to add new keywords to the parser. These new keywords can either be mapped to existing tokens or simply ignored. For example, recent versions of the GNU compiler recognize the keywords `__signed__` and `__extension__`. The first one obviously is a synonym for `signed`, while the second one is only a marker for a language extension.

Using the preprocessor, you could of course do the following:

```
$c->Define(qw( __signed__=signed __extension__= ));
```

However, the preprocessor symbols could be undefined or redefined in the code, and

```
#ifdef __signed__
# undef __signed__
#endif

typedef __extension__ __signed__ long long s_quad;
```

would generate a parse error, because `__signed__` is an unexpected identifier.

Instead of utilizing the preprocessor, you'll have to create mappings for the new keywords directly in the parser using `KeywordMap`. In the above example, you want to map `__signed__` to the built-in C keyword `signed` and ignore `__extension__`. This could be done with the following code:

```
$c->KeywordMap({ __signed__    => 'signed',
                 __extension__ => undef });
```

You can specify any valid identifier as hash key, and either a valid C keyword or `undef` as hash value. Having configured the object that way, you could parse even

```
#ifdef __signed__
# undef __signed__
#endif

typedef __extension__ __signed__ long long s_quad;
```

without problems.

Note that `KeywordMap` and `DisabledKeywords` perfectly work together. You could, for example, disable the `signed` keyword, but still have `__signed__` mapped to the original `signed` token:

```
$c->configure(DisabledKeywords => [ 'signed' ],
              KeywordMap       => { __signed__  => 'signed' });
```

This would allow you to define

```
typedef __signed__ long signed;
```

which would normally be a syntax error because `signed` cannot be used as an identifier.

`UnsignedChars` => **0** | **1**

Use this boolean option if you want characters to be unsigned if specified without an explicit `signed` or `unsigned` type specifier. By default, characters are signed.

`UnsignedBitfields` => **0** | **1**

Use this boolean option if you want bitfields to be unsigned if specified without an explicit `signed` or `unsigned` type specifier. By default, bitfields are signed.

`Warnings` => **0** | **1**

Use this boolean option if you want warnings to be issued during the parsing of source code. Currently, warnings are only reported by the preprocessor, so don't expect the output to cover everything.

By default, warnings are turned off and only errors will be reported. However, even these errors are turned off if you run without the `-w` flag.

`HasCPPComments` => **0** | **1**

Use this option to turn C++ comments on or off. By default, C++ comments are enabled. Disabling C++ comments may be necessary if your code includes strange things like:

```
one = 4 //* <- divide */ 4;
two = 2;
```

With C++ comments, the above will be interpreted as

```
one = 4
two = 2;
```

which will obviously be a syntax error, but without C++ comments, it will be interpreted as

```
one = 4 / 4;
two = 2;
```

which is correct.

`HasMacroVAARGS` => **0** | **1**

Use this option to turn the `__VA_ARGS__` macro expansion on or off. If this is enabled (which is the default), you can use variable length argument lists in your preprocessor macros.

```
#define DEBUG( ... )  fprintf( stderr, __VA_ARGS__ )
```

There's normally no reason to turn that feature off.

`Include` => [**INCLUDES**]

Use this option to set the include path for the internal preprocessor. The option value is a reference to an array of strings, each string holding a directory that should be searched for includes.

`Define` => [**DEFINES**]

Use this option to define symbols in the preprocessor. The option value is, again, a reference to an array of strings. Each string can be either just a symbol or an assignment to a symbol. This is completely equivalent to what the `-D` option does for most preprocessors.

The following will define the symbol `FOO` and define `BAR` to be `12345`:

```
$c->configure(Define => [qw( FOO BAR=12345 )]);
```

`Assert` => [**ASSERTIONS**]

Use this option to make assertions in the preprocessor. If you don't know what assertions are, don't be concerned, since they're deprecated anyway. They are, however, used in some system's include files. The value is an array reference, just like for the macro definitions. Only the way the assertions are defined is a bit different and mimics the way they are defined with the `#assert` directive:

```
$c->configure(Assert => ['foo(bar)']);
```

`OrderMembers` => **0** | **1**

When using unpack on compounds and iterating over the returned hash, the order of the compound members is generally not preserved due to the nature of hash tables. It is not even guaranteed that the order is the same between different runs of the same program. This can be very annoying if you simply use to dump your data structures and the compound members always show up in a different order.

By setting `OrderMembers` to a non-zero value, all hashes returned by unpack are tied to a class that preserves the order of the hash keys. This way, all compound members will be returned in the correct order just as they are defined in your C code.

```
use Convert::Binary::C;
use Data::Dumper;

$c = Convert::Binary::C->new->parse(<<'ENDC');
struct test {
  char one;
  char two;
  struct {
    char never;
```

```
    char change;
    char this;
    char order;
  } three;
  char four;
};
ENDC

$data = "Convert";

$u1 = $c->unpack('test', $data);
$c->OrderMembers(1);
$u2 = $c->unpack('test', $data);

print Data::Dumper->Dump([$u1, $u2], [qw(u1 u2)]);
```

This will print something like:

```
$u1 = {
  'three' => {
    'change' => 118,
    'order' => 114,
    'this' => 101,
    'never' => 110
  },
  'one' => 67,
  'two' => 111,
  'four' => 116
};
$u2 = {
  'one' => 67,
  'two' => 111,
  'three' => {
    'never' => 110,
    'change' => 118,
    'this' => 101,
    'order' => 114
  },
  'four' => 116
};
```

To be able to use this option, you have to install either the Tie::Hash::Indexed or the Tie::IxHash module. If both are installed, Convert::Binary::C will give preference to Tie::Hash::Indexed because it's faster.

When using this option, you should keep in mind that tied hashes are significantly slower and consume more memory than ordinary hashes, even when the class they're tied to is implemented efficiently. So don't turn this option on if you don't have to.

You can also influence hash member ordering by using the CBC_ORDER_MEMBERS environment variable.

`Bitfields` => **{ OPTION => VALUE, ... }**

Use this option to specify and configure a bitfield layouting engine. You can choose an engine by passing its name to the `Engine` option, like:

```
$c->configure(Bitfields => { Engine => 'Generic' });
```

Each engine can have its own set of options, although currently none of them does.

You can choose between the following bitfield engines:

`Generic`
: This engine implements the behaviour of most UNIX C compilers, including GCC. It does not handle packed bitfields yet.

`Microsoft`
: This engine implements the behaviour of Microsoft's `cl` compiler. It should be fairly complete and can handle packed bitfields.

`Simple`
: This engine is only used for testing the bitfield infrastructure in Convert::Binary::C. There's usually no reason to use it.

You can reconfigure all options even after you have parsed some code. The changes will be applied to the already parsed definitions. This works as long as array lengths are not affected by the changes. If you have Alignment and IntSize set to 4 and parse code like this

```
typedef struct {
  char abc;
  int  day;
} foo;

struct bar {
  foo  zap[2*sizeof(foo)];
};
```

the array `zap` in `struct bar` will obviously have 16 elements. If you reconfigure the alignment to 1 now, the size of `foo` is now 5 instead of 8. While the alignment is adjusted correctly, the number of elements in array `zap` will still be 16 and will not be changed to 10.

1.5.3 parse

`parse` CODE

Parses a string of valid C code. All enumeration, compound and type definitions are extracted. You can call the `parse` and `parse_file` methods as often as you like to add further definitions to the Convert::Binary::C object.

`parse` will throw an exception if an error occurs. On success, the method returns a reference to its object.

See the section on Parsing C code on page 12 for an example.

1.5.4 parse_file

`parse_file` FILE

Parses a C source file. All enumeration, compound and type definitions are extracted. You can call the `parse` and `parse_file` methods as often as you like to add further definitions to the Convert::Binary::C object.

`parse_file` will search the include path given via the `Include` option for the file if it cannot find it in the current directory.

`parse_file` will throw an exception if an error occurs. On success, the method returns a reference to its object.

See the section on Parsing C code on page 12 for an example.

When calling parse or parse_file multiple times, you may use types previously defined, but you are not allowed to redefine types. The state of the preprocessor is also saved, so you may also use defines from a previous parse. This works only as long as the preprocessor is not reset. See the section on Preprocessor configuration on page 13 for details.

When you're parsing C source files instead of C header files, note that local definitions are ignored. This means that type definitions hidden within functions will not be recognized by Convert::Binary::C. This is necessary because different functions (even different blocks within the same function) can define types with the same name:

```
void my_func(int i)
{
  if (i < 10)
  {
    enum digit { ONE, TWO, THREE } x = ONE;
    printf("%d, %d\n", i, x);
  }
  else
  {
    enum digit { THREE, TWO, ONE } x = ONE;
    printf("%d, %d\n", i, x);
  }
}
```

The above is a valid piece of C code, but it's not possible for Convert::Binary::C to distinguish between the different definitions of `enum digit`, as they're only defined locally within the corresponding block.

1.5.5 clean

`clean`

Clears all information that has been collected during previous calls to parse or parse_file. You can use this method if you want to parse some entirely different code, but with the same configuration.

The clean method returns a reference to its object.

1.5.6 clone

`clone`

Makes the object return an exact independent copy of itself.

```
$c = new Convert::Binary::C Include => ['/usr/include'];
$c->parse_file('definitions.c');
$clone = $c->clone;
```

The above code is technically equivalent (Mostly. Actually, using sourcify and parse might alter the order of the parsed data, which would make methods such as compound return the definitions in a different order.) to:

```
$c = new Convert::Binary::C Include => ['/usr/include'];
$c->parse_file('definitions.c');
$clone = new Convert::Binary::C %{$c->configure};
$clone->parse($c->sourcify);
```

Using clone is just a lot faster.

1.5.7 def

`def` **NAME**

`def` **TYPE**

If you need to know if a definition for a certain type name exists, use this method. You pass it the name of an enum, struct, union or typedef, and it will return a non-empty string being either `"enum"`, `"struct"`, `"union"`, or `"typedef"` if there's a definition for the type in question, an empty string if there's no such definition, or `undef` if the name is completely unknown. If the type can be interpreted as a basic type, `"basic"` will be returned.

If you pass in a TYPE, the output will be slightly different. If the specified member exists, the `def` method will return `"member"`. If the member doesn't exist, or if the type cannot have members, the empty string will be returned. Again, if the name of the type is completely unknown, `undef` will be returned. This may be useful if you want to check if a certain member exists within a compound, for example.

```
use Convert::Binary::C;

my $c = Convert::Binary::C->new->parse(<<'ENDC');

typedef struct __not  not;
typedef struct __not *ptr;

struct foo {
  enum bar *xxx;
};

typedef int quad[4];

ENDC

for my $type (qw( not ptr foo bar xxx foo.xxx foo.abc xxx.yyy
                  quad quad[3] quad[5] quad[-3] short[1] ),
              'unsigned long')
{
  my $def = $c->def($type);
  printf "%-14s  =>  %s\n",
          $type,     defined $def ? "'$def'" : 'undef';
}
```

The following would be returned by the `def` method:

```
not             =>  ''
ptr             =>  'typedef'
foo             =>  'struct'
bar             =>  ''
xxx             =>  undef
foo.xxx         =>  'member'
foo.abc         =>  ''
xxx.yyy         =>  undef
quad            =>  'typedef'
quad[3]         =>  'member'
quad[5]         =>  'member'
quad[-3]        =>  'member'
short[1]        =>  undef
unsigned long   =>  'basic'
```

So, if def returns a non-empty string, you can safely use any other method with that type's name or with that member expression.

Concerning arrays, note that the index into an array doesn't need to be within the bounds of the array's definition, just like in C. In the above example, quad[5] and quad[-3] are valid members of the quad array, even though it is declared to have only four elements.

In cases where the typedef namespace overlaps with the namespace of enums/structs/unions, the def method will give preference to the typedef and will thus return the string "typedef". You could however force interpretation as an enum, struct or union by putting "enum", "struct" or "union" in front of the type's name.

1.5.8 defined

defined **MACRO**

You can use the defined method to find out if a certain macro is defined, just like you would use the defined operator of the preprocessor. For example, the following code

```
use Convert::Binary::C;

my $c = Convert::Binary::C->new->parse(<<'ENDC');

#define ADD(a, b) ((a) + (b))

#if 1
# define DEFINED
#else
# define UNDEFINED
#endif

ENDC

for my $macro (qw( ADD DEFINED UNDEFINED )) {
  my $not = $c->defined($macro) ? '' : ' not';
  print "Macro '$macro' is$not defined.\n";
}
```

would print:

```
Macro 'ADD' is defined.
Macro 'DEFINED' is defined.
Macro 'UNDEFINED' is not defined.
```

You have to keep in mind that this works only as long as the preprocessor is not reset. See the section on Preprocessor configuration on page 13 for details.

1.5.9 pack

pack **TYPE**

pack **TYPE, DATA**

pack TYPE, DATA, STRING

Use this method to pack a complex data structure into a binary string according to a type definition that has been previously parsed. DATA must be a scalar matching the type definition. C structures and unions are represented by references to Perl hashes, C arrays by references to Perl arrays.

```
use Convert::Binary::C;
use Data::Dumper;
use Data::Hexdumper;

$c = Convert::Binary::C->new( ByteOrder => 'BigEndian'
                            , LongSize  => 4
                            , ShortSize => 2
                            )
                       ->parse(<<'ENDC');
struct test {
  char    ary[3];
  union {
    short word[2];
    long  quad;
  }       uni;
};
ENDC
```

Hashes don't have to contain a key for each compound member and arrays may be truncated:

```
$binary = $c->pack('test', { ary => [1, 2], uni => { quad => 42 } });
```

Elements not defined in the Perl data structure will be set to zero in the packed byte string. If you pass `undef` as or simply omit the second parameter, the whole string will be initialized with zero bytes. On success, the packed byte string is returned.

```
print hexdump(data => $binary);
```

The above code would print:

```
  0x0000 : 01 02 00 00 00 00 2A                             : ......*
```

You could also use unpack and dump the data structure.

```
$unpacked = $c->unpack('test', $binary);
print Data::Dumper->Dump([$unpacked], ['unpacked']);
```

This would print:

```
$unpacked = {
  'uni' => {
    'word' => [
      0,
      42
    ],
    'quad' => 42
  },
  'ary' => [
```

```
    1,
    2,
    0
  ]
};
```

If TYPE refers to a compound object, you may pack any member of that compound object. Simply add a member expression to the type name, just as you would access the member in C:

```
$array = $c->pack('test.ary', [1, 2, 3]);
print hexdump(data => $array);

$value = $c->pack('test.uni.word[1]', 2);
print hexdump(data => $value);
```

This would give you:

```
  0x0000 : 01 02 03                                       : ...
  0x0000 : 00 02                                          : ..
```

Call pack with the optional STRING argument if you want to use an existing binary string to insert the data. If called in a void context, pack will directly modify the string you passed as the third argument. Otherwise, a copy of the string is created, and pack will modify and return the copy, so the original string will remain unchanged.

The 3-argument version may be useful if you want to change only a few members of a complex data structure without having to unpack everything, change the members, and then pack again (which could waste lots of memory and CPU cycles). So, instead of doing something like

```
$test = $c->unpack('test', $binary);
$test->{uni}{quad} = 4711;
$new = $c->pack('test', $test);
```

to change the `uni.quad` member of `$packed`, you could simply do either

```
$new = $c->pack('test', { uni => { quad => 4711 } }, $binary);
```

or

```
$c->pack('test', { uni => { quad => 4711 } }, $binary);
```

while the latter would directly modify `$packed`. Besides this code being a lot shorter (and perhaps even more readable), it can be significantly faster if you're dealing with really big data blocks.

If the length of the input string is less than the size required by the type, the string (or its copy) is extended and the extended part is initialized to zero. If the length is more than the size required by the type, the string is kept at that length, and also a copy would be an exact copy of that string.

```
$too_short = pack "C*", (1 .. 4);
$too_long  = pack "C*", (1 .. 20);

$c->pack('test', { uni => { quad => 0x4711 } }, $too_short);
print "too_short:\n", hexdump(data => $too_short);
```

```
$copy = $c->pack('test', { uni => { quad => 0x4711 } }, $too_long);
print "\ncopy:\n", hexdump(data => $copy);
```

This would print:

```
too_short:
  0x0000 : 01 02 03 00 00 47 11                           : .....G.

copy:
  0x0000 : 01 02 03 00 00 47 11 08 09 0A 0B 0C 0D 0E 0F 10 : .....G..........
  0x0010 : 11 12 13 14                                    : ....
```

1.5.10 unpack

`unpack` **TYPE, STRING**

Use this method to unpack a binary string and create an arbitrarily complex Perl data structure based on a previously parsed type definition.

```
use Convert::Binary::C;
use Data::Dumper;

$c = Convert::Binary::C->new( ByteOrder => 'BigEndian'
                            , LongSize  => 4
                            , ShortSize => 2
                            )
                      ->parse( <<'ENDC' );
struct test {
  char    ary[3];
  union {
    short word[2];
    long *quad;
  }       uni;
};
ENDC

# Generate some binary dummy data
$binary = pack "C*", 1 .. $c->sizeof('test');
```

On failure, e.g. if the specified type cannot be found, the method will throw an exception. On success, a reference to a complex Perl data structure is returned, which can directly be dumped using the Data::Dumper module:

```
$unpacked = $c->unpack('test', $binary);
print Dumper($unpacked);
```

This would print:

```
$VAR1 = {
  'uni' => {
    'word' => [
      1029,
      1543
```

```
    ],
    'quad' => 67438087
  },
  'ary' => [
    1,
    2,
    3
  ]
};
```

If TYPE refers to a compound object, you may unpack any member of that compound object. Simply add a member expression to the type name, just as you would access the member in C:

```
$binary2 = substr $binary, $c->offsetof('test', 'uni.word');

$unpack1 = $unpacked->{uni}{word};
$unpack2 = $c->unpack('test.uni.word', $binary2);

print Data::Dumper->Dump([$unpack1, $unpack2], [qw(unpack1 unpack2)]);
```

You will find that the output is exactly the same for both `$unpack1` and `$unpack2`:

```
$unpack1 = [
  1029,
  1543
];
$unpack2 = [
  1029,
  1543
];
```

When `unpack` is called in list context, it will unpack as many elements as possible from STRING, including zero if STRING is not long enough.

1.5.11 initializer

`initializer` **TYPE**

`initializer` **TYPE, DATA**

The `initializer` method can be used retrieve an initializer string for a certain TYPE. This can be useful if you have to initialize only a couple of members in a huge compound type or if you simply want to generate initializers automatically.

```
struct date {
  unsigned year : 12;
  unsigned month:  4;
  unsigned day  :  5;
  unsigned hour :  5;
  unsigned min  :  6;
};
```

```
typedef struct {
  enum { DATE, QWORD } type;
  short number;
  union {
    struct date   date;
    unsigned long qword;
  } choice;
} data;
```

Given the above code has been parsed

```
$init = $c->initializer('data');
print "data x = $init;\n";
```

would print the following:

```
data x = {
      0,
      0,
      {
              {
                      0,
                      0,
                      0,
                      0,
                      0
              }
      }
};
```

You could directly put that into a C program, although it probably isn't very useful yet. It becomes more useful if you actually specify how you want to initialize the type:

```
$data = {
  type   => 'QWORD',
  choice => {
    date  => { month => 12, day => 24 },
    qword => 4711,
  },
  stuff => 'yes?',
};

$init = $c->initializer('data', $data);
print "data x = $init;\n";
```

This would print the following:

```
data x = {
      QWORD,
      0,
      {
              {
                      0,
```

```
                    12,
                    24,
                    0,
                    0
                }
        }
};
```

As only the first member of a `union` can be initialized, `choice.qword` is ignored. You will not be warned about the fact that you probably tried to initialize a member other than the first. This is considered a feature, because it allows you to use unpack to generate the initializer data:

```
$data = $c->unpack('data', $binary);
$init = $c->initializer('data', $data);
```

Since unpack unpacks all union members, you would otherwise have to delete all but the first one previous to feeding it into initializer.

Also, `stuff` is ignored, because it actually isn't a member of `data`. You won't be warned about that either.

1.5.12 sizeof

`sizeof` **TYPE**

This method will return the size of a C type in bytes. If it cannot find the type, it will throw an exception.

If the type defines some kind of compound object, you may ask for the size of a member of that compound object:

```
$size = $c->sizeof('test.uni.word[1]');
```

This would set `$size` to 2.

1.5.13 typeof

`typeof` **TYPE**

This method will return the type of a C member. While this only makes sense for compound types, it's legal to also use it for non-compound types. If it cannot find the type, it will throw an exception.

The typeof method can be used on any valid member, even on arrays or unnamed types. It will always return a string that holds the name (or in case of unnamed types only the class) of the type, optionally followed by a `'*'` character to indicate it's a pointer type, and optionally followed by one or more array dimensions if it's an array type. If the type is a bitfield, the type name is followed by a colon and the number of bits.

```
struct test {
  char    ary[3];
  union {
    short word[2];
    long *quad;
  }       uni;
  struct {
```

```
    unsigned short six:6;
    unsigned short ten:10;
  }       bits;
};
```

Given the above C code has been parsed, calls to typeof would return the following values:

```
$c->typeof('test')             => 'struct test'
$c->typeof('test.ary')         => 'char [3]'
$c->typeof('test.uni')         => 'union'
$c->typeof('test.uni.quad')    => 'long *'
$c->typeof('test.uni.word')    => 'short [2]'
$c->typeof('test.uni.word[1]') => 'short'
$c->typeof('test.bits')        => 'struct'
$c->typeof('test.bits.six')    => 'unsigned short :6'
$c->typeof('test.bits.ten')    => 'unsigned short :10'
```

1.5.14 offsetof

`offsetof` **TYPE, MEMBER**

You can use offsetof just like the C macro of same denominator. It will simply return the offset (in bytes) of MEMBER relative to TYPE.

```
use Convert::Binary::C;

$c = Convert::Binary::C->new( Alignment   => 4
                            , LongSize    => 4
                            , PointerSize => 4
                            )
                       ->parse(<<'ENDC');
typedef struct {
  char abc;
  long day;
  int *ptr;
} week;

struct test {
  week zap[8];
};
ENDC

@args = (
  ['test',        'zap[5].day'  ],
  ['test.zap[2]', 'day'         ],
  ['test',        'zap[5].day+1'],
  ['test',        'zap[-3].ptr' ],
);

for (@args) {
  my $offset = eval { $c->offsetof(@$_) };
  printf "\$c->offsetof('%s', '%s') => $offset\n", @$_;
}
```

The final loop will print:

```
$c->offsetof('test', 'zap[5].day') => 64
$c->offsetof('test.zap[2]', 'day') => 4
$c->offsetof('test', 'zap[5].day+1') => 65
$c->offsetof('test', 'zap[-3].ptr') => -28
```

- The first iteration simply shows that the offset of `zap[5].day` is 64 relative to the beginning of `struct test`.
- You may additionally specify a member for the type passed as the first argument, as shown in the second iteration.
- The offset suffix is also supported by `offsetof`, so the third iteration will correctly print 65.
- The last iteration demonstrates that even out-of-bounds array indices are handled correctly, just as they are handled in C.

Unlike the C macro, `offsetof` also works on array types.

```
$offset = $c->offsetof('test.zap', '[3].ptr+2');
print "offset = $offset";
```

This will print:

```
offset = 46
```

If TYPE is a compound, MEMBER may optionally be prefixed with a dot, so

```
printf "offset = %d\n", $c->offsetof('week', 'day');
printf "offset = %d\n", $c->offsetof('week', '.day');
```

are both equivalent and will print

```
offset = 4
offset = 4
```

This allows to

- use the C macro style, without a leading dot, and
- directly use the output of the `member` method, which includes a leading dot for compound types, as input for the MEMBER argument.

1.5.15 member

`member` **TYPE**

`member` **TYPE, OFFSET**

You can think of `member` as being the reverse of the `offsetof` method. However, as this is more complex, there's no equivalent to `member` in the C language.

Usually this method is used if you want to retrieve the name of the member that is located at a specific offset of a previously parsed type.

```
use Convert::Binary::C;
```

```
$c = Convert::Binary::C->new( Alignment   => 4
                            , LongSize    => 4
                            , PointerSize => 4
                            )
                       ->parse(<<'ENDC');
typedef struct {
  char abc;
  long day;
  int *ptr;
} week;

struct test {
  week zap[8];
};
ENDC

for my $offset (24, 39, 69, 99) {
  print "\$c->member('test', $offset)";
  my $member = eval { $c->member('test', $offset) };
  print $@ ? "\n  exception: $@" : " => '$member'\n";
}
```

This will print:

```
$c->member('test', 24) => '.zap[2].abc'
$c->member('test', 39) => '.zap[3]+3'
$c->member('test', 69) => '.zap[5].ptr+1'
$c->member('test', 99)
  exception: Offset 99 out of range (0 <= offset < 96)
```

- The output of the first iteration is obvious. The member `zap[2].abc` is located at offset 24 of `struct test`.
- In the second iteration, the offset points into a region of padding bytes and thus no member of `week` can be named. Instead of a member name the offset relative to `zap[3]` is appended.
- In the third iteration, the offset points to `zap[5].ptr`. However, `zap[5].ptr` is located at 68, not at 69, and thus the remaining offset of 1 is also appended.
- The last iteration causes an exception because the offset of 99 is not valid for `struct test` since the size of `struct test` is only 96. You might argue that this is inconsistent, since `offsetof` can also handle out-of-bounds array members. But as soon as you have more than one level of array nesting, there's an infinite number of out-of-bounds members for a single given offset, so it would be impossible to return a list of all members.

You can additionally specify a member for the type passed as the first argument:

```
$member = $c->member('test.zap[2]', 6);
print $member;
```

This will print:

```
.day+2
```

Like `offsetof`, `member` also works on array types:

```
$member = $c->member('test.zap', 42);
print $member;
```

This will print:

```
[3].day+2
```

While the behaviour for `structs` is quite obvious, the behaviour for `unions` is rather tricky. As a single offset usually references more than one member of a union, there are certain rules that the algorithm uses for determining the *best* member.

- The first non-compound member that is referenced without an offset has the highest priority.
- If no member is referenced without an offset, the first non-compound member that is referenced with an offset will be returned.
- Otherwise the first padding region that is encountered will be taken.

As an example, given 4-byte-alignment and the union

```
union choice {
  struct {
    char  color[2];
    long  size;
    char  taste;
  }       apple;
  char    grape[3];
  struct {
    long  weight;
    short price[3];
  }       melon;
};
```

the `member` method would return what is shown in the *Member* column of the following table. The *Type* column shows the result of the `typeof` method when passing the corresponding member.

```
Offset   Member               Type
--------------------------------------
   0     .apple.color[0]      'char'
   1     .apple.color[1]      'char'
   2     .grape[2]            'char'
   3     .melon.weight+3      'long'
   4     .apple.size          'long'
   5     .apple.size+1        'long'
   6     .melon.price[1]      'short'
   7     .apple.size+3        'long'
   8     .apple.taste         'char'
   9     .melon.price[2]+1    'short'
  10     .apple+10            'struct'
  11     .apple+11            'struct'
```

It's like having a stack of all the union members and looking through the stack for the shiniest piece you can see. The beginning of a member (denoted by uppercase letters) is always shinier than the rest of a member, while padding regions (denoted by dashes) aren't shiny at all.

```
Offset   0   1   2   3   4   5   6   7   8   9  10  11
-----------------------------------------------------
apple   (C) (C)  -   -  (S) (s)  s  (s) (T)  -  (-) (-)
grape    G   G  (G)
melon    W   w   w  (w)  P   p  (P)  p   P  (p)  -   -
```

If you look through that stack from top to bottom, you'll end up at the parenthesized members.

Alternatively, if you're not only interested in the *best* member, you can call `member` in list context, which makes it return *all* members referenced by the given offset.

```
Offset   Member                Type
--------------------------------------
   0     .apple.color[0]       'char'
         .grape[0]             'char'
         .melon.weight         'long'
   1     .apple.color[1]       'char'
         .grape[1]             'char'
         .melon.weight+1       'long'
   2     .grape[2]             'char'
         .melon.weight+2       'long'
         .apple+2              'struct'
   3     .melon.weight+3       'long'
         .apple+3              'struct'
   4     .apple.size           'long'
         .melon.price[0]       'short'
   5     .apple.size+1         'long'
         .melon.price[0]+1     'short'
   6     .melon.price[1]       'short'
         .apple.size+2         'long'
   7     .apple.size+3         'long'
         .melon.price[1]+1     'short'
   8     .apple.taste          'char'
         .melon.price[2]       'short'
   9     .melon.price[2]+1     'short'
         .apple+9              'struct'
  10     .apple+10             'struct'
         .melon+10             'struct'
  11     .apple+11             'struct'
         .melon+11             'struct'
```

The first member returned is always the *best* member. The other members are sorted according to the rules given above. This means that members referenced without an offset are followed by members referenced with an offset. Padding regions will be at the end.

If OFFSET is not given in the method call, `member` will return a list of *all* possible members of TYPE.

```
print "$_\n" for $c->member('choice');
```

This will print:

```
.apple.color[0]
.apple.color[1]
.apple.size
```

```
.apple.taste
.grape[0]
.grape[1]
.grape[2]
.melon.weight
.melon.price[0]
.melon.price[1]
.melon.price[2]
```

In scalar context, the number of possible members is returned.

1.5.16 tag

`tag` **TYPE**

`tag` **TYPE, TAG**

`tag` **TYPE, TAG1 => VALUE1, TAG2 => VALUE2, ...**

The `tag` method can be used to tag properties to a TYPE. It's a bit like having `configure` for individual types.

See the section on Using Tags on page 18 for an example.

Note that while you can tag whole types as well as compound members, it is not possible to tag array members, i.e. you cannot treat, for example, `a[1]` and `a[2]` differently.

Also note that in code like this

```
struct test {
  int a;
  struct {
    int x;
  } b, c;
};
```

if you tag `test.b.x`, this will also tag `test.c.x` implicitly.

It is also possible to tag basic types if you really want to do that, for example:

```
$c->tag('int', Format => 'Binary');
```

To remove a tag from a type, you can either set that tag to `undef`, for example

```
$c->tag('test', Hooks => undef);
```

or use `untag`.

To see if a tag is attached to a type or to get the value of a tag, pass only the type and tag name to `tag`:

```
$c->tag('test.a', Format => 'Binary');

$hooks = $c->tag('test.a', 'Hooks');
$format = $c->tag('test.a', 'Format');
```

This will give you:

```
$hooks = undef;
$format = 'Binary';
```

To see which tags are attached to a type, pass only the type. The tag method will now return a hash reference containing all tags attached to the type:

```
$tags = $c->tag('test.a');
```

This will give you:

```
$tags = {
  'Format' => 'Binary'
};
```

tag will throw an exception if an error occurs. If called as a 'set' method, it will return a reference to its object, allowing you to chain together consecutive method calls.

Note that when a compound is inlined, tags attached to the inlined compound are ignored, for example:

```
$c->parse(<<ENDC);
struct header {
  int id;
  int len;
  unsigned flags;
};

struct message {
  struct header;
  short samples[32];
};
ENDC

for my $type (qw( header message header.len )) {
  $c->tag($type, Hooks => { unpack => sub { print "unpack: $type\n"; @_ } });
}

for my $type (qw( header message )) {
  print "[unpacking $type]\n";
  $u = $c->unpack($type, $data);
}
```

This will print:

```
[unpacking header]
unpack: header.len
unpack: header
[unpacking message]
unpack: header.len
unpack: message
```

As you can see from the above output, tags attached to members of inlined compounds (`header.len` are still handled.

The following tags can be configured:

`Format` => **'Binary'** | **'String'**

The `Format` tag allows you to control the way binary data is converted by `pack` and `unpack`.

If you tag a `TYPE` as `Binary`, it will not be converted at all, i.e. it will be passed through as a binary string.

If you tag it as `String`, it will be treated like a null-terminated C string, i.e. `unpack` will convert the C string to a Perl string and vice versa.

See the section on The Format Tag on page 18 for an example.

`ByteOrder` => **'BigEndian'** | **'LittleEndian'**

The `ByteOrder` tag allows you to explicitly set the byte order of a TYPE.

See the section on The Byteorder Tag on page 20 for an example.

`Dimension` => **'*'**

`Dimension` => **VALUE**

`Dimension` => **MEMBER**

`Dimension` => **SUB**

`Dimension` => [**SUB, ARGS**]

The `Dimension` tag allows you to alter the size of an array dynamically.

You can tag fixed size arrays as being flexible using `'*'`. This is useful if you cannot use flexible array members in your source code.

```
$c->tag('type.array', Dimension => '*');
```

You can also tag an array to have a fixed size different from the one it was originally declared with.

```
$c->tag('type.array', Dimension => 42);
```

If the array is a member of a compound, you can also tag it with to have a size corresponding to the value of another member in that compound.

```
$c->tag('type.array', Dimension => 'count');
```

Finally, you can specify a subroutine that is called when the size of the array needs to be determined.

```
$c->tag('type.array', Dimension => \&get_count);
```

By default, and if the array is a compound member, that subroutine will be passed a reference to the hash storing the data for the compound.

You can also instruct Convert::Binary::C to pass additional arguments to the subroutine by passing an array reference instead of the subroutine reference. This array contains the subroutine reference as well as a list of arguments. It is possible to define certain special arguments using the `arg` method.

```
$c->tag('type.array', Dimension => [\&get_count, $c->arg('SELF'), 42]);
```

See the section on The Dimension Tag on page 22 for various examples.

`Hooks` => **{ HOOK => SUB, HOOK => [SUB, ARGS], ... }, ...**

The `Hooks` tag allows you to register subroutines as hooks.

Hooks are called whenever a certain `TYPE` is packed or unpacked. Hooks are currently considered an **experimental** feature.

`HOOK` can be one of the following:

```
pack
unpack
pack_ptr
unpack_ptr
```

pack and unpack hooks are called when processing their TYPE, while pack_ptr and unpack_ptr hooks are called when processing pointers to their TYPE.

SUB is a reference to a subroutine that usually takes one input argument, processes it and returns one output argument.

Alternatively, you can pass a custom list of arguments to the hook by using an array reference instead of SUB that holds the subroutine reference in the first element and the arguments to be passed to the subroutine as the other elements. This way, you can even pass special arguments to the hook using the arg method.

Here are a few examples for registering hooks:

```
$c->tag('ObjectType', Hooks => {
          pack   => \&obj_pack,
          unpack => \&obj_unpack
        });

$c->tag('ProtocolId', Hooks => {
          unpack => sub { $protos[$_[0]] }
        });

$c->tag('ProtocolId', Hooks => {
          unpack_ptr => [sub {
                           sprintf "$_[0]:{0x%X}", $_[1]
                         },
                         $c->arg('TYPE', 'DATA')
                        ],
        });
```

Note that the above example registers both an unpack hook and an unpack_ptr hook for ProtocolId with two separate calls to tag. As long as you don't explicitly overwrite a previously registered hook, it won't be modified or removed by registering other hooks for the same TYPE.

To remove all registered hooks for a type, simply remove the Hooks tag:

```
$c->untag('ProtocolId', 'Hooks');
```

To remove only a single hook, pass undef as SUB instead of a subroutine reference:

```
$c->tag('ObjectType', Hooks => { pack => undef });
```

If all hooks are removed, the whole Hooks tag is removed.

See the section on The Hooks Tag on page 26 for examples on how to use hooks.

1.5.17 untag

untag **TYPE**

untag **TYPE, TAG1, TAG2, ...**

Use the untag method to remove one, more, or all tags from a type. If you don't pass any tag names, all tags attached to the type will be removed. Otherwise only the listed tags will be removed.

See the section on Using Tags on page 18 for an example.

1.5.18 arg

`arg` **'ARG', ...**

Creates placeholders for special arguments to be passed to hooks or other subroutines. These arguments are currently:

`SELF`

A reference to the calling Convert::Binary::C object. This may be useful if you need to work with the object inside the subroutine.

`TYPE`

The name of the type that is currently being processed by the hook.

`DATA`

The data argument that is passed to the subroutine.

`HOOK`

The type of the hook as which the subroutine has been called, for example `pack` or `unpack_ptr`.

`arg` will return a placeholder for each argument it is being passed. Note that not all arguments may be supported depending on the context of the subroutine.

1.5.19 dependencies

`dependencies`

After some code has been parsed using either the `parse` or `parse_file` methods, the `dependencies` method can be used to retrieve information about all files that the object depends on, i.e. all files that have been parsed.

In scalar context, the method returns a hash reference. Each key is the name of a file. The values are again hash references, each of which holds the size, modification time (mtime), and change time (ctime) of the file at the moment it was parsed.

```
use Convert::Binary::C;
use Data::Dumper;

#------------------------------------------------------------
# Create object, set include path, parse 'string.h' header
#------------------------------------------------------------
my $c = Convert::Binary::C->new
        ->Include('/usr/lib/gcc-lib/i686-pc-linux-gnu/3.3.6/include',
                  '/usr/include')
        ->parse_file('string.h');

#------------------------------------------------------------
# Get dependencies of the object, extract dependency files
#------------------------------------------------------------
my $depend = $c->dependencies;
my @files  = keys %$depend;

#-----------------------------
# Dump dependencies and files
#-----------------------------
print Data::Dumper->Dump([$depend, \@files],
                         [qw( depend   *files )]);
```

The above code would print something like this:

```
$depend = {
  '/usr/include/features.h' => {
    'ctime' => 1180783212,
    'mtime' => 1180783206,
    'size' => 11734
  },
  '/usr/include/gnu/stubs-32.h' => {
    'ctime' => 1180783211,
    'mtime' => 1180783206,
    'size' => 647
  },
  '/usr/include/sys/cdefs.h' => {
    'ctime' => 1180783211,
    'mtime' => 1180783206,
    'size' => 11339
  },
  '/usr/include/gnu/stubs.h' => {
    'ctime' => 1180783211,
    'mtime' => 1180783206,
    'size' => 315
  },
  '/usr/include/string.h' => {
    'ctime' => 1180783212,
    'mtime' => 1180783206,
    'size' => 16281
  },
  '/usr/lib/gcc-lib/i686-pc-linux-gnu/3.3.6/include/stddef.h' => {
    'ctime' => 1167821862,
    'mtime' => 1167821861,
    'size' => 12695
  },
  '/usr/include/bits/wordsize.h' => {
    'ctime' => 1180783211,
    'mtime' => 1180783206,
    'size' => 873
  }
};
@files = (
  '/usr/include/features.h',
  '/usr/include/gnu/stubs-32.h',
  '/usr/include/sys/cdefs.h',
  '/usr/include/gnu/stubs.h',
  '/usr/include/string.h',
  '/usr/lib/gcc-lib/i686-pc-linux-gnu/3.3.6/include/stddef.h',
  '/usr/include/bits/wordsize.h'
);
```

In list context, the method returns the names of all files that have been parsed, i.e. the following lines are equivalent:

```
@files = keys %{$c->dependencies};
```

```
@files = $c->dependencies;
```

1.5.20 sourcify

`sourcify`

`sourcify` **CONFIG**

Returns a string that holds the C source code necessary to represent all parsed C data structures.

```
use Convert::Binary::C;

$c = new Convert::Binary::C;
$c->parse(<<'END');

#define ADD(a, b) ((a) + (b))
#define NUMBER 42

typedef struct _mytype mytype;

struct _mytype {
  union {
    int         iCount;
    enum count *pCount;
  } counter;
#pragma pack( push, 1 )
  struct {
    char string[NUMBER];
    int  array[NUMBER/sizeof(int)];
  } storage;
#pragma pack( pop )
  mytype *next;
};

enum count { ZERO, ONE, TWO, THREE };

END

print $c->sourcify;
```

The above code would print something like this:

```
/* typedef predeclarations */

typedef struct _mytype mytype;

/* defined enums */

enum count
{
      ZERO,
      ONE,
      TWO,
      THREE
};
```

```
/* defined structs and unions */

struct _mytype
{
      union
      {
              int iCount;
              enum count *pCount;
      } counter;
#pragma pack(push, 1)
      struct
      {
              char string[42];
              int array[10];
      } storage;
#pragma pack(pop)
      mytype *next;
};
```

The purpose of the sourcify method is to enable some kind of platform-independent caching. The C code generated by sourcify can be parsed by any standard C compiler, as well as of course by the Convert::Binary::C parser. However, the code may be significantly shorter than the code that has originally been parsed.

When parsing a typical header file, it's easily possible that you need to open dozens of other files that are included from that file, and end up parsing several hundred kilobytes of C code. Since most of it is usually preprocessor directives, function prototypes and comments, the sourcify function strips this down to a few kilobytes. Saving the sourcify string and parsing it next time instead of the original code may be a lot faster.

The sourcify method takes a hash reference as an optional argument. It can be used to tweak the method's output. The following options can be configured.

`Context` => **0** | **1**

Turns preprocessor context information on or off. If this is turned on, sourcify will insert `#line` preprocessor directives in its output. So in the above example

```
print $c->sourcify({ Context => 1 });
```

would print:

```
/* typedef predeclarations */

typedef struct _mytype mytype;

/* defined enums */

#line 21 "[buffer]"
enum count
{
      ZERO,
      ONE,
      TWO,
      THREE
};

/* defined structs and unions */
```

```
#line 7 "[buffer]"
struct _mytype
{
#line 8 "[buffer]"
      union
      {
              int iCount;
              enum count *pCount;
      } counter;
#pragma pack(push, 1)
#line 13 "[buffer]"
      struct
      {
              char string[42];
              int array[10];
      } storage;
#pragma pack(pop)
      mytype *next;
};
```

Note that `"[buffer]"` refers to the here-doc buffer when using `parse`.

`Defines` => **0** | **1**

Turn this on if you want all the defined macros to be part of the source code output. Given the example code above

```
print $c->sourcify({ Defines => 1 });
```

would print:

```
/* typedef predeclarations */

typedef struct _mytype mytype;

/* defined enums */

enum count
{
      ZERO,
      ONE,
      TWO,
      THREE
};

/* defined structs and unions */

struct _mytype
{
      union
      {
              int iCount;
              enum count *pCount;
      } counter;
#pragma pack(push, 1)
      struct
      {
              char string[42];
              int array[10];
```

```
      } storage;
#pragma pack(pop)
      mytype *next;
};

/* preprocessor defines */

#define ADD(a, b) ((a) + (b))
#define NUMBER 42
```

The macro definitions always appear at the end of the source code. The order of the macro definitions is undefined.

The following methods can be used to retrieve information about the definitions that have been parsed. The examples given in the description for `enum`, `compound` and `typedef` all assume this piece of C code has been parsed:

```
#define ABC_SIZE 2
#define MULTIPLY(x, y) ((x)*(y))

#ifdef ABC_SIZE
# define DEFINED
#else
# define NOT_DEFINED
#endif

typedef unsigned long U32;
typedef void *any;

enum __socket_type
{
  SOCK_STREAM    = 1,
  SOCK_DGRAM     = 2,
  SOCK_RAW       = 3,
  SOCK_RDM       = 4,
  SOCK_SEQPACKET = 5,
  SOCK_PACKET    = 10
};

struct STRUCT_SV {
  void *sv_any;
  U32   sv_refcnt;
  U32   sv_flags;
};

typedef union {
  int abc[ABC_SIZE];
  struct xxx {
    int a;
    int b;
  }   ab[3][4];
  any ptr;
} test;
```

1.5.21 enum_names

`enum_names`

Returns a list of identifiers of all defined enumeration objects. Enumeration objects don't necessarily have an identifier, so something like

```
enum { A, B, C };
```

will obviously not appear in the list returned by the `enum_names` method. Also, enumerations that are not defined within the source code - like in

```
struct foo {
  enum weekday *pWeekday;
  unsigned long year;
};
```

where only a pointer to the `weekday` enumeration object is used - will not be returned, even though they have an identifier. So for the above two enumerations, `enum_names` will return an empty list:

```
@names = $c->enum_names;
```

The only way to retrieve a list of all enumeration identifiers is to use the `enum` method without additional arguments. You can get a list of all enumeration objects that have an identifier by using

```
@enums = map { $_->{identifier} || () } $c->enum;
```

but these may not have a definition. Thus, the two arrays would look like this:

```
@names = ();
@enums = ('weekday');
```

The `def` method returns a true value for all identifiers returned by `enum_names`.

1.5.22 enum

enum

`enum` **LIST**

Returns a list of references to hashes containing detailed information about all enumerations that have been parsed.

If a list of enumeration identifiers is passed to the method, the returned list will only contain hash references for those enumerations. The enumeration identifiers may optionally be prefixed by `enum`.

If an enumeration identifier cannot be found, the returned list will contain an undefined value at that position.

In scalar context, the number of enumerations will be returned as long as the number of arguments to the method call is not 1. In the latter case, a hash reference holding information for the enumeration will be returned.

The list returned by the `enum` method looks similar to this:

```
@enum = (
  {
    'enumerators' => {
      'SOCK_STREAM' => 1,
      'SOCK_RAW' => 3,
      'SOCK_SEQPACKET' => 5,
      'SOCK_RDM' => 4,
      'SOCK_PACKET' => 10,
      'SOCK_DGRAM' => 2
    },
    'identifier' => '__socket_type',
    'context' => 'definitions.c(13)',
    'size' => 4,
    'sign' => 0
  }
);
```

`identifier`
: holds the enumeration identifier. This key is not present if the enumeration has no identifier.

`context`
: is the context in which the enumeration is defined. This is the filename followed by the line number in parentheses.

`enumerators`
: is a reference to a hash table that holds all enumerators of the enumeration.

`sign`
: is a boolean indicating if the enumeration is signed (i.e. has negative values).

One useful application may be to create a hash table that holds all enumerators of all defined enumerations:

```
%enum = map %{ $_->{enumerators} || {} }, $c->enum;
```

The `%enum` hash table would then be:

```
%enum = (
  'SOCK_STREAM' => 1,
  'SOCK_RAW' => 3,
  'SOCK_SEQPACKET' => 5,
  'SOCK_RDM' => 4,
  'SOCK_DGRAM' => 2,
  'SOCK_PACKET' => 10
);
```

1.5.23 compound_names

`compound_names`

Returns a list of identifiers of all structs and unions (compound data structures) that are defined in the parsed source code. Like enumerations, compounds don't need to have an identifier, nor do they need to be defined.

Again, the only way to retrieve information about all struct and union objects is to use the `compound` method and don't pass it any arguments. If you should need a list of all struct and union identifiers, you can use:

```
@compound = map { $_->{identifier} || () } $c->compound;
```

The def method returns a true value for all identifiers returned by compound_names.

If you need the names of only the structs or only the unions, use the struct_names and union_names methods respectively.

1.5.24 compound

`compound`

`compound` **LIST**

Returns a list of references to hashes containing detailed information about all compounds (structs and unions) that have been parsed.

If a list of struct/union identifiers is passed to the method, the returned list will only contain hash references for those compounds. The identifiers may optionally be prefixed by `struct` or `union`, which limits the search to the specified kind of compound.

If an identifier cannot be found, the returned list will contain an undefined value at that position.

In scalar context, the number of compounds will be returned as long as the number of arguments to the method call is not 1. In the latter case, a hash reference holding information for the compound will be returned.

The list returned by the compound method looks similar to this:

```
@compound = (
  {
    'identifier' => 'STRUCT_SV',
    'align' => 1,
    'context' => 'definitions.c(23)',
    'pack' => 0,
    'type' => 'struct',
    'declarations' => [
      {
        'declarators' => [
          {
            'declarator' => '*sv_any',
            'size' => 4,
            'offset' => 0
          }
        ],
        'type' => 'void'
      },
      {
        'declarators' => [
          {
            'declarator' => 'sv_refcnt',
            'size' => 4,
            'offset' => 4
          }
        ],
        'type' => 'U32'
      },
```

```
    {
      'declarators' => [
        {
          'declarator' => 'sv_flags',
          'size' => 4,
          'offset' => 8
        }
      ],
      'type' => 'U32'
    }
  ],
  'size' => 12
},
{
  'identifier' => 'xxx',
  'align' => 1,
  'context' => 'definitions.c(31)',
  'pack' => 0,
  'type' => 'struct',
  'declarations' => [
    {
      'declarators' => [
        {
          'declarator' => 'a',
          'size' => 4,
          'offset' => 0
        }
      ],
      'type' => 'int'
    },
    {
      'declarators' => [
        {
          'declarator' => 'b',
          'size' => 4,
          'offset' => 4
        }
      ],
      'type' => 'int'
    }
  ],
  'size' => 8
},
{
  'align' => 1,
  'context' => 'definitions.c(29)',
  'pack' => 0,
  'type' => 'union',
  'declarations' => [
    {
      'declarators' => [
        {
```

```
            'declarator' => 'abc[2]',
            'size' => 8,
            'offset' => 0
          }
        ],
        'type' => 'int'
      },
      {
        'declarators' => [
          {
            'declarator' => 'ab[3][4]',
            'size' => 96,
            'offset' => 0
          }
        ],
        'type' => 'struct xxx'
      },
      {
        'declarators' => [
          {
            'declarator' => 'ptr',
            'size' => 4,
            'offset' => 0
          }
        ],
        'type' => 'any'
      }
    ],
    'size' => 96
  }
);
```

`identifier`
: holds the struct or union identifier. This key is not present if the compound has no identifier.

`context`
: is the context in which the struct or union is defined. This is the filename followed by the line number in parentheses.

`type`
: is either 'struct' or 'union'.

`size`
: is the size of the struct or union.

`align`
: is the alignment of the struct or union.

`pack`
: is the struct member alignment if the compound is packed, or zero otherwise.

`declarations`
: is an array of hash references describing each struct declaration:

  `type`
  : is the type of the struct declaration. This may be a string or a reference to a hash describing the type.

`declarators`
: is an array of hashes describing each declarator:

  `declarator`
  : is a string representation of the declarator.

  `offset`
  : is the offset of the struct member represented by the current declarator relative to the beginning of the struct or union.

  `size`
  : is the size occupied by the struct member represented by the current declarator.

It may be useful to have separate lists for structs and unions. One way to retrieve such lists would be to use

```
push @{$_->{type} eq 'union' ? \@unions : \@structs}, $_
    for $c->compound;
```

However, you should use the `struct` and `union` methods, which is a lot simpler:

```
@structs = $c->struct;
@unions  = $c->union;
```

1.5.25 struct_names

`struct_names`

Returns a list of all defined struct identifiers. This is equivalent to calling `compound_names`, just that it only returns the names of the struct identifiers and doesn't return the names of the union identifiers.

1.5.26 struct

`struct`

`struct` **LIST**

Like the `compound` method, but only allows for structs.

1.5.27 union_names

`union_names`

Returns a list of all defined union identifiers. This is equivalent to calling `compound_names`, just that it only returns the names of the union identifiers and doesn't return the names of the struct identifiers.

1.5.28 union

`union`

`union` **LIST**

Like the `compound` method, but only allows for unions.

1.5.29 typedef_names

`typedef_names`

Returns a list of all defined typedef identifiers. Typedefs that do not specify a type that you could actually work with will not be returned.

The `def` method returns a true value for all identifiers returned by `typedef_names`.

1.5.30 typedef

`typedef`

`typedef` **LIST**

Returns a list of references to hashes containing detailed information about all typedefs that have been parsed.

If a list of typedef identifiers is passed to the method, the returned list will only contain hash references for those typedefs.

If an identifier cannot be found, the returned list will contain an undefined value at that position.

In scalar context, the number of typedefs will be returned as long as the number of arguments to the method call is not 1. In the latter case, a hash reference holding information for the typedef will be returned.

The list returned by the `typedef` method looks similar to this:

```
@typedef = (
  {
    'declarator' => 'U32',
    'type' => 'unsigned long'
  },
  {
    'declarator' => '*any',
    'type' => 'void'
  },
  {
    'declarator' => 'test',
    'type' => {
      'align' => 1,
      'context' => 'definitions.c(29)',
      'pack' => 0,
      'type' => 'union',
      'declarations' => [
        {
          'declarators' => [
            {
              'declarator' => 'abc[2]',
              'size' => 8,
              'offset' => 0
            }
          ],
          'type' => 'int'
        },
        {
```

```
            'declarators' => [
              {
                'declarator' => 'ab[3][4]',
                'size' => 96,
                'offset' => 0
              }
            ],
            'type' => 'struct xxx'
          },
          {
            'declarators' => [
              {
                'declarator' => 'ptr',
                'size' => 4,
                'offset' => 0
              }
            ],
            'type' => 'any'
          }
        ],
        'size' => 96
      }
    }
  );
```

`declarator`
: is the type declarator.

`type`
: is the type specification. This may be a string or a reference to a hash describing the type. See `enum` and `compound` for a description on how to interpret this hash.

1.5.31 macro_names

`macro_names`

Returns a list of all defined macro names.

The list returned by the `macro_names` method looks similar to this:

```
  @macro_names = (
    '__STDC_VERSION__',
    '__STDC_HOSTED__',
    'DEFINED',
    'MULTIPLY',
    'ABC_SIZE'
  );
```

This works only as long as the preprocessor is not reset. See the section on Preprocessor configuration on page 13 for details.

1.5.32 macro

`macro`

`macro` **LIST**

Returns the definitions for all defined macros.

If a list of macro names is passed to the method, the returned list will only contain the definitions for those macros. For undefined macros, `undef` will be returned.

The list returned by the `macro` method looks similar to this:

```
@macro = (
  '__STDC_VERSION__ 199901L',
  '__STDC_HOSTED__ 1',
  'DEFINED',
  'MULTIPLY(x, y) ((x)*(y))',
  'ABC_SIZE 2'
);
```

This works only as long as the preprocessor is not reset. See the section on Preprocessor configuration on page 13 for details.

1.6 Functions

You can alternatively call the following functions as methods on Convert::Binary::C objects.

1.6.1 feature

`feature` **STRING**

Checks if Convert::Binary::C was built with certain features. For example,

```
print "debugging version"
    if Convert::Binary::C::feature('debug');
```

will check if Convert::Binary::C was built with debugging support enabled. The `feature` function returns `1` if the feature is enabled, `0` if the feature is disabled, and `undef` if the feature is unknown. Currently the only features that can be checked are `ieeefp` and `debug`.

You can enable or disable certain features at compile time of the module by using the

```
perl Makefile.PL enable-feature disable-feature
```

syntax.

1.6.2 native

`native`

`native` **STRING**

Returns the value of a property of the native system that Convert::Binary::C was built on. For example,

```
$size = Convert::Binary::C::native('IntSize');
```

will fetch the size of an `int` on the native system. The following properties can be queried:

```
Alignment
ByteOrder
CharSize
CompoundAlignment
DoubleSize
EnumSize
FloatSize
IntSize
LongDoubleSize
LongLongSize
LongSize
PointerSize
ShortSize
UnsignedBitfields
UnsignedChars
```

You can also call `native` without arguments, in which case it will return a reference to a hash with all properties, like:

```
$native = {
  'ByteOrder' => 'LittleEndian',
  'LongSize' => 4,
  'IntSize' => 4,
  'ShortSize' => 2,
  'FloatSize' => 4,
  'Alignment' => 4,
  'LongLongSize' => 8,
  'LongDoubleSize' => 12,
  'UnsignedChars' => 0,
  'DoubleSize' => 8,
  'CharSize' => 1,
  'UnsignedBitfields' => 0,
  'EnumSize' => 4,
  'CompoundAlignment' => 1,
  'PointerSize' => 4
};
```

The contents of that hash are suitable for passing them to the `configure` method.

1.7 Debugging

Like perl itself, Convert::Binary::C can be compiled with debugging support that can then be selectively enabled at runtime. You can specify whether you like to build Convert::Binary::C with debugging support or not by explicitly giving an argument to *Makefile.PL*. Use

```
perl Makefile.PL enable-debug
```

to enable debugging, or

```
perl Makefile.PL disable-debug
```

to disable debugging. The default will depend on how your perl binary was built. If it was built with `-DDEBUGGING`, Convert::Binary::C will be built with debugging support, too.

Once you have built Convert::Binary::C with debugging support, you can use the following syntax to enable debug output. Instead of

```
use Convert::Binary::C;
```

you simply say

```
use Convert::Binary::C debug => 'all';
```

which will enable all debug output. However, I don't recommend to enable all debug output, because that can be a fairly large amount.

1.7.1 Debugging options

Instead of saying `all`, you can pass a string that consists of one or more of the following characters:

```
m   enable memory allocation tracing
M   enable memory allocation & assertion tracing

h   enable hash table debugging
H   enable hash table dumps

d   enable debug output from the XS module
c   enable debug output from the ctlib
t   enable debug output about type objects

l   enable debug output from the C lexer
p   enable debug output from the C parser
P   enable debug output from the C preprocessor
r   enable debug output from the #pragma parser

y   enable debug output from yacc (bison)
```

So the following might give you a brief overview of what's going on inside Convert::Binary::C:

```
use Convert::Binary::C debug => 'dct';
```

When you want to debug memory allocation using

```
use Convert::Binary::C debug => 'm';
```

you can use the Perl script *check˙alloc.pl* that resides in the *ctlib/util/tool* directory to extract statistics about memory usage and information about memory leaks from the resulting debug output.

1.7.2 Redirecting debug output

By default, all debug output is written to `stderr`. You can, however, redirect the debug output to a file with the `debugfile` option:

```
use Convert::Binary::C debug     => 'dcthHm',
                       debugfile => './debug.out';
```

If the file cannot be opened, you'll receive a warning and the output will go the `stderr` way again.

Alternatively, you can use the environment variables CBC_DEBUG_OPT and CBC_DEBUG_FILE to turn on debug output.

If Convert::Binary::C is built without debugging support, passing the `debug` or `debugfile` options will cause a warning to be issued. The corresponding environment variables will simply be ignored.

1.8 Environment

1.8.1 CBC_ORDER_MEMBERS

Setting this variable to a non-zero value will globally turn on hash key ordering for compound members. Have a look at the `OrderMembers` option for details.

Setting the variable to the name of a perl module will additionally use this module instead of the predefined modules for member ordering to tie the hashes to.

1.8.2 CBC_DEBUG_OPT

If Convert::Binary::C is built with debugging support, you can use this variable to specify the debugging options.

1.8.3 CBC_DEBUG_FILE

If Convert::Binary::C is built with debugging support, you can use this variable to redirect the debug output to a file.

1.8.4 CBC_DISABLE_PARSER

This variable is intended purely for development. Setting it to a non-zero value disables the Convert::Binary::C parser, which means that no information is collected from the file or code that is parsed. However, the preprocessor will run, which is useful for benchmarking the preprocessor.

1.9 Flexible Array Members And Incomplete Types

Flexible array members are a feature introduced with ISO-C99. It's a common problem that you have a variable length data field at the end of a structure, for example an array of characters at the end of a message struct. ISO-C99 allows you to write this as:

```
struct message {
  long header;
  char data[];
};
```

The advantage is that you clearly indicate that the size of the appended data is variable, and that the `data` member doesn't contribute to the size of the `message` structure.

When packing or unpacking data, Convert::Binary::C deals with flexible array members as if their length was adjustable. For example, `unpack` will adapt the length of the array depending on the input string:

```
$msg1 = $c->unpack('message', 'abcdefg');
$msg2 = $c->unpack('message', 'abcdefghijkl');
```

The following data is unpacked:

```
$msg1 = {
  'data' => [
    101,
    102,
    103
  ],
  'header' => 1633837924
};
$msg2 = {
  'data' => [
    101,
    102,
    103,
    104,
    105,
    106,
    107,
    108
  ],
  'header' => 1633837924
};
```

Similarly, pack will adjust the length of the output string according to the data you feed in:

```
use Data::Hexdumper;

$msg = {
  header => 4711,
  data   => [0x10, 0x20, 0x30, 0x40, 0x77..0x88],
};

$data = $c->pack('message', $msg);

print hexdump(data => $data);
```

This would print:

```
  0x0000 : 00 00 12 67 10 20 30 40 77 78 79 7A 7B 7C 7D 7E : ...g..0@wxyz{|}~
  0x0010 : 7F 80 81 82 83 84 85 86 87 88                   : ..........
```

Incomplete types such as

```
typedef unsigned long array[];
```

are handled in exactly the same way. Thus, you can easily

```
$array = $c->unpack('array', '?'x20);
```

which will unpack the following array:

```
$array = [
  1061109567,
  1061109567,
  1061109567,
  1061109567,
  1061109567
];
```

You can also alter the length of an array using the `Dimension` tag.

1.10 Floating Point Values

When using Convert::Binary::C to handle floating point values, you have to be aware of some limitations.

You're usually safe if all your platforms are using the IEEE floating point format. During the Convert::Binary::C build process, the `ieeefp` feature will automatically be enabled if the host is using IEEE floating point. You can check for this feature at runtime using the `feature` function:

```
if (Convert::Binary::C::feature('ieeefp')) {
  # do something
}
```

When IEEE floating point support is enabled, the module can also handle floating point values of a different byteorder.

If your host platform is not using IEEE floating point, the `ieeefp` feature will be disabled. Convert::Binary::C then will be more restrictive, refusing to handle any non-native floating point values.

However, Convert::Binary::C cannot detect the floating point format used by your target platform. It can only try to prevent problems in obvious cases. If you know your target platform has a completely different floating point format, don't use floating point conversion at all.

Whenever Convert::Binary::C detects that it cannot properly do floating point value conversion, it will issue a warning and will not attempt to convert the floating point value.

1.11 Bitfields

Bitfield support in Convert::Binary::C is currently in an **experimental** state. You are encouraged to test it, but you should not blindly rely on its results.

You are also encouraged to supply layouting algorithms for compilers whose bitfield implementation is not handled correctly at the moment. Even better that the plain algorithm is of course a patch that adds a new bitfield layouting engine.

While bitfields may not be handled correctly by the conversion routines yet, they are always parsed correctly. This means that you can reliably use the declarator fields as returned by the `struct` or `typedef` methods. Given the following source

```
struct bitfield {
  int seven:7;
  int :1;
  int four:4, :0;
  int integer;
};
```

a call to `struct` will return

```
@struct = (
  {
    'identifier' => 'bitfield',
    'align' => 1,
    'context' => 'bitfields.c(1)',
    'pack' => 0,
    'type' => 'struct',
    'declarations' => [
      {
        'declarators' => [
          {
            'declarator' => 'seven:7'
          }
        ],
        'type' => 'int'
      },
      {
        'declarators' => [
          {
            'declarator' => ':1'
          }
        ],
        'type' => 'int'
      },
      {
        'declarators' => [
          {
            'declarator' => 'four:4'
          },
          {
            'declarator' => ':0'
```

```
          }
        ],
        'type' => 'int'
      },
      {
        'declarators' => [
          {
            'declarator' => 'integer',
            'size' => 4,
            'offset' => 4
          }
        ],
        'type' => 'int'
      }
    ],
    'size' => 8
  }
);
```

No size/offset keys will currently be returned for bitfield entries.

1.12 Multithreading

Convert::Binary::C was designed to be thread-safe.

1.13 Inheritance

If you wish to derive a new class from Convert::Binary::C, this is relatively easy. Despite their XS implementation, Convert::Binary::C objects are actually blessed hash references.

The XS data is stored in a read-only hash value for the key that is the empty string. So it is safe to use any non-empty hash key when deriving your own class. In addition, Convert::Binary::C does quite a lot of checks to detect corruption in the object hash.

If you store private data in the hash, you should override the `clone` method and provide the necessary code to clone your private data. You'll have to call `SUPER::clone`, but this will only clone the Convert::Binary::C part of the object.

For an example of a derived class, you can have a look at Convert::Binary::C::Cached.

1.14 Portability

Convert::Binary::C should build and run on most of the platforms that Perl runs on:

- Various Linux systems
- Various BSD systems
- HP-UX

- Compaq/HP Tru64 Unix
- Mac-OS X
- Cygwin
- Windows 98/NT/2000/XP

Also, many architectures are supported:

- Various Intel Pentium and Itanium systems
- Various Alpha systems
- HP PA-RISC
- Power-PC
- StrongARM

The module should build with any perl binary from 5.004 up to the latest development version.

1.15 Comparison With Similar Modules

Most of the time when you're really looking for Convert::Binary::C you'll actually end up finding one of the following modules. Some of them have different goals, so it's probably worth pointing out the differences.

1.15.1 C::Include

Like Convert::Binary::C, this module aims at doing conversion from and to binary data based on C types. However, its configurability is very limited compared to Convert::Binary::C. Also, it does not parse all C code correctly. It's slower than Convert::Binary::C, doesn't have a preprocessor. On the plus side, it's written in pure Perl.

1.15.2 C::DynaLib::Struct

This module doesn't allow you to reuse your C source code. One main goal of Convert::Binary::C was to avoid code duplication or, even worse, having to maintain different representations of your data structures. Like C::Include, C::DynaLib::Struct is rather limited in its configurability.

1.15.3 Win32::API::Struct

This module has a special purpose. It aims at building structs for interfacing Perl code with Windows API code.

1.16 Credits

- My love Jennifer for always being there, for filling my life with joy and last but not least for proofreading the documentation.
- Alain Barbet <alian@cpan.org> for testing and debugging support.
- Mitchell N. Charity for giving me pointers into various interesting directions.
- Alexis Denis for making me improve (externally) and simplify (internally) floating point support. He can also be blamed (indirectly) for the `initializer` method, as I need it in my effort to support bitfields some day.
- Michael J. Hohmann <mjh@scientist.de> for endless discussions on our way to and back home from work, and for making me think about supporting `pack` and `unpack` for compound members.
- Thorsten Jens <thojens@gmx.de> for testing the package on various platforms.
- Mark Overmeer <mark@overmeer.net> for suggesting the module name and giving invaluable feedback.
- Thomas Pornin <pornin@bolet.org> for his excellent `ucpp` preprocessor library.
- Marc Rosenthal for his suggestions and support.
- James Roskind, as his C parser was a great starting point to fix all the problems I had with my original parser based only on the ANSI ruleset.
- Gisbert W. Selke for spotting some interesting bugs and providing extensive reports.
- Steffen Zimmermann for a prolific discussion on the cloning algorithm.

1.17 Mailing List

There's also a mailing list that you can join:

```
convert-binary-c@yahoogroups.com
```

To subscribe, simply send mail to:

```
convert-binary-c-subscribe@yahoogroups.com
```

You can use this mailing list for non-bug problems, questions or discussions.

1.18 Bugs

I'm sure there are still lots of bugs in the code for this module. If you find any bugs, Convert::Binary::C doesn't seem to build on your system or any of its tests fail, please use the CPAN Request Tracker at http://rt.cpan.org/ to create a ticket for the module. Alternatively, just send a mail to <mhx@cpan.org>.

1.19 Experimental Features

Some features in Convert::Binary::C are marked as experimental. This has most probably one of the following reasons:

- The feature does not behave in exactly the way that I wish it did, possibly due to some limitations in the current design of the module.
- The feature hasn't been tested enough and may completely fail to produce the expected results.

I hope to fix most issues with these experimental features someday, but this may mean that I have to change the way they currently work in a way that's not backwards compatible. So if any of these features is useful to you, you can use it, but you should be aware that the behaviour or the interface may change in future releases of this module.

1.20 Todo

If you're interested in what I currently plan to improve (or fix), have a look at the *TODO* file.

1.21 Postcards

If you're using my module and like it, you can show your appreciation by sending me a postcard from where you live. I won't urge you to do it, it's completely up to you. To me, this is just a very nice way of receiving feedback about my work. Please send your postcard to:

```
Marcus Holland-Moritz
Kuppinger Weg 28
71116 Gaertringen
GERMANY
```

If you feel that sending a postcard is too much effort, you maybe want to rate the module at http://cpanratings.perl.org/.

1.22 Copyright

1.23 See Also

See the *ccconfig* manpage, the *perl* manpage, the *perldata* manpage, the *perlop* manpage, the *perlvar* manpage, the *Data::Dumper* manpage and the *Scalar::Util* manpage.

Convert::Binary::C::Cached

Caching for Convert::Binary::C

2.1 Synopsis

```
use Convert::Binary::C::Cached;
use Data::Dumper;

#------------------------
# Create a cached object
#------------------------
$c = Convert::Binary::C::Cached->new(
       Cache   => '/tmp/cache.c',
       Include => [
         '/usr/lib/gcc-lib/i686-pc-linux-gnu/3.3.6/include',
         '/usr/include',
       ],
     );

#-----------------------------------------------------
# Parse 'time.h' and dump the definition of timespec
#-----------------------------------------------------
$c->parse_file('time.h');

print Dumper($c->struct('timespec'));
```

2.2 Description

Convert::Binary::C::Cached simply adds caching capability to Convert::Binary::C. You can use it in just the same way that you would use Convert::Binary::C. The interface is exactly the same.

To use the caching capability, you must pass the `Cache` option to the constructor. If you don't pass it, you will receive an ordinary Convert::Binary::C object. The argument to the `Cache` option is the file that is used for caching this object.

The caching algorithm automatically detects when the cache file cannot be used and the original code has to be parsed. In that case, the cache file is updated. An update of the cache file can be triggered by one or more of the following factors:

- The cache file doesn't exist, which is obvious.

- The cache file is corrupt, i.e. cannot be parsed.
- The object's configuration has changed.
- The embedded code for a `parse` method call has changed.
- At least one of the files that the object depends on does not exist or has a different size or a different modification or change timestamp.

2.3 Limitations

You cannot call `parse` or `parse_file` more that once when using a Convert::Binary::C::Cached object. This isn't a big problem, as you usually don't call them multiple times.

If a dependency file changes, but the change affects neither the size nor the timestamps of that file, the caching algorithm cannot detect that an update is required.

2.4 Copyright

2.5 See Also

See the *Convert::Binary::C* manpage.

ccconfig

Get Convert::Binary::C configuration for a compiler

3.1 Synopsis

ccconfig *options* [– compiler-options]

options:

```
-c
--cc              compiler   compiler executable to test
                             default: auto-determined

-o
--output-file     file       output filename
                             default: output to stdout

-f
--output-format   format     output format
                             default: dumper

--basename        name       basename of the temporary test files
                             default: _t_e_s_t

-I
--inc-path        path       manually set compiler include path

--preprocess      rule       compiler rule for preprocessing
--compile-obj     rule       compiler rule for compiling objects
--compile-exe     rule       compiler rule for compiling executables

--c-ext           ext        extension of C source files
--pp-ext          ext        extension of preprocessor output files
--obj-ext         ext        extension of object files
--exe-ext         ext        extension of executable files

--nodelete                   don't delete temporary files
--norun                      don't try to run executables
--quiet                      don't display anything
--nostatus                   don't display status indicator
```

```
--version                   print version number

--debug                     debug mode
```

Placeholders allowed in compiler rules:

```
%c    C source file
%o    object file
%e    executable file
%i    preprocessor output file
|     result is written to stdout (only at end of rule)
```

3.2 Description

`ccconfig` will try to determine a usable configuration for Convert::Binary::C from testing a compiler executable. It is not necessary that the binaries generated by the compiler can be executed, so `ccconfig` can also be used for cross-compilers.

This tool is still experimental, and you should neither rely on its output without checking, nor expect it to work in your environment.

3.3 Options

3.3.1 `--cc` compiler

This option allows you to explicitly specify a compiler executable. This is especially useful if you don't want to use your system compiler. If this options is not given, `ccconfig` tries to guess a compiler.

3.3.2 `--output-file` file

Write Convert::Binary::C configuration to the specified file. The default is to write the configuration to `stdout`.

3.3.3 `--output-format` format

Specify the output format of the Convert::Binary::C configuration. The following formats are currently supported:

```
dumper      Output a %config hash using Data::Dumper
require     Output in a format suitable for require
```

The default is `dumper`.

3.3.4 `--basename` name

Allows you to change the base name of the temporary test files. This is used along with the various `-ext` options to build the filenames of C source files, preprocessor output files, object files and executables.

3.3.5 `--inc-path` path

This option allows you to manually set the include path of the compiler. This is useful if `ccconfig` cannot determine the include path automatically, most probably because it cannot parse the preprocessor output. This option can be specified more than once.

3.3.6 `--preprocess` rule

Using this option, you can specify a *rule* that `ccconfig` uses to run the compiler to get preprocessor output. Most compilers write the preprocessor output to standard output when given the `-E` option, i.e.

```
cc -E foo.c
```

will preprocess *foo.c* to standard output. The corresponding rule for `ccconfig` would be:

```
ccconfig --preprocess='-E %c |'
```

The <%c> will be replaced with the C source filename, and the pipe symbol signals that the result will be written to standard output.

The following placeholders can be used in `ccconfig` rules:

```
%c    C source file
%o    object file
%e    executable file
%i    preprocessor output file
```

Usually, `ccconfig` tries to figure out the correct rules on its own.

3.3.7 `--compile-obj` rule

Like `--preprocess`, this option allows you to define a rule for how to compile an object file. For most compilers, this rule will be something like

```
ccconfig --compile-obj='-c -o %o %c'
```

3.3.8 `--compile-exe` rule

Like `--preprocess`, this option allows you to define a rule for how to compile an executable file. For most compilers, this rule will be something like

```
ccconfig --compile-exe='-o %e %c'
```

Note that it is sufficient to specify either `--compile-obj` or `--compile-exe`. So if your compiler can only create object files, that's just fine.

3.3.9 `--c-ext`

This option is used along with `--basename` to build the name of a C source file. This is usually set to `.c`.

3.3.10 --pp-ext

This option is used along with `--basename` to build the name of a preprocessor output file.

3.3.11 --obj-ext

This option is used along with `--basename` to build the name of an object file.

3.3.12 --exe-ext

This option is used along with `--basename` to build the name of an executable file.

3.3.13 --nodelete

Don't attempt to delete temporary files that have been created by the compiler. Normally, `ccconfig` will look for all files with the same basename as the temporary test file and delete them.

3.3.14 --norun

You can specify this option if the executables generated by your compiler cannot be run on your machine, i.e. if you have a cross-compiler. However, `ccconfig` will automatically find out that it cannot run the executables.

When this option is set, a different set of algorithms is used to determine a couple of configuration settings. These algorithms are all based upon placing a special signature in the object file. They are less reliable that the standard algorithms, so you shouldn't use them unless you have to.

3.3.15 --quiet

Don't display anything except for the final configuration.

3.3.16 --nostatus

Hide the status indicator. Recommended if you want to redirect the script output to a file:

```
ccconfig --nostatus >config.pl 2>ccconfig.log
```

3.3.17 --version

Writes the program name, version and path to standard output.

3.3.18 --debug

Generate tons of debug output. Don't use unless you know what you're doing.

3.4 Examples

Normally, a simple

```
ccconfig
```

without arguments is enough if you want the configuration for your system compiler. While `ccconfig` is running, it will write lots of status information to `stderr`. When it's done, it will usually dump a Perl hash table to `stdout` which can be directly used as a configuration for Convert::Binary::C.

If you want the configuration for a different compiler, or `ccconfig` cannot determine your system compiler automatically, use

```
ccconfig -c gcc32
```

if your compiler's name is `gcc32`.

If you want to pass additional options to the compiler, you can do so after a double-dash on the command line:

```
ccconfig -- -g -DDEBUGGING
```

or

```
ccconfig -c gcc32 -- -ansi -fshort-enums
```

If you'd like to interface with the Perl core, you may find a suitable configuration using something like:

```
ccconfig --cc=‘perl -MConfig -e 'print $Config{cc}'‘ \
         -- ‘perl -MConfig -e 'print $Config{ccflags}'‘
```

3.5 Copyright

3.6 See Also

See the *Convert::Binary::C* manpage.