Thin-C

Thin-C is a way of writing C that makes it

- harder for programmers to make mistakes
- easier to detect defects
- easier to auto-generate

Make types obvious

`typedef` is free, so use it freely. There are very few occasions when a plain `int` (or any other basic type) is called for. Most of the time the `int` is actually representing some dimensional quantity, such as number of seconds, number of bytes, height, width, area, speed, ratio, iteration count, or some sort of flag, status, error number, file descriptor, comparison result, character number, signal number, etc. If this is the case then name the type accordingly.

```
typedef int ExitStatus;
typedef unsigned int WordCount;
typedef int Comparison;
typedef double LengthInMetres;
```

One benefit of this is that function prototypes become more readable. The following prototype …

```
int control_shared_memory(int, int, struct shmid_ds *);
```

… becomes …

```
SharedMemoryIndex control_shared_memory(
    SharedMemoryId,
    SharedMemoryCommand,
    struct shmid_ds *);
```

Even if the original prototype had included parameter names …

```
int control_shared_memory(int shmid, int cmd, struct shmid_ds
*buf);
```

… there is no way from the prototype alone to determine what the return type may contain.

Make types harder to misuse

Even when using `typedef` it is still possible to use a variable of that type in the wrong way. For example, given these type definitions and function declaration…

```
typedef int Hour;
typedef int Minute;

Minute hhmm_to_Minute(Hour, Minute);
```

it is still possible to call the function incorrectly …

```
    Minute a = 23;
    Hour b = 11;
    Minute since_midnight = hhmm_to_Minute(a,b);
```

and it is also possible to use the variables inappropriately, e.g.

```
    x = a * b;  // multiply hours by minutes!!
```

The way to prevent this misuse is to wrap the declaration in a `struct`:

```
typedef struct { int hidden; } Hour;
typedef struct { int hidden; } Minute;
```

now we will get a compile-time error when we try this …

```
    Minute a = {53};
    Hour b = {11};
    Minute since_midnight = hhmm_to_Minute(a,b);
```

and also when we use almost any operator on the wrapped type …

```
    // compile-time errors:
    a * b;
    a = b;
    a = (Minute)b;
    a++;
    if (a) ;
```

It is important to realise that a `struct` wrapping an `int` has the same size, same alignment and same *address* as the wrapped `int`. As far as the object code is concerned the wrapping `struct` is invisible, it behaves as if the `int` was not wrapped. But from a source code point of view it is a vital way of ensuring type safety.

```
#include <stdio.h>
struct wrapping_s { int wrapped; };
int main()
{
    struct wrapping_s wrapping = { 0 };
    void * wrapping_addr = (void *)&wrapping;
    void * wrapped_addr = (void *)&(wrapping.wrapped);

    printf("wrapped:  size=%lu address=%p\n",
        sizeof(wrapping.wrapped), wrapped_addr);
    printf("wrapping: size=%lu address=%p\n",
        sizeof(wrapping), wrapping_addr);

    return 0;
}
```

Make strong types immutable "by default"

(In Thin-C a struct type wrapping a single member as in the examples above is known as a *strong type* and we will use this term from now on in preference to *wrapped type* or *wrapping type*.)
One of the unfortunate historical accidents that befell C was that the concept of *constness* wasn't added to the language until the C89/C90 standard. This means that, by default, types are non-const (mutable, changeable) and the programmer has to remember to add the `const` keyword to ensure type immutability. In Thin-C, strong types should have a `const` member by default; if we need a non-const version of the strong type then we declare a different type whose name reflects the fact that the type is mutable (in this article we use the `Sink` suffix to imply that you can put something in the type. NB You can use any suffix (*e.g.* `_Mutable`) as long as you do it consistently).

```
typedef struct { int const hidden; } Hour;
typedef struct { int hidden; } HourSink;
```

However, the programmer still has to remember to put the `const` keyword in the type definition. So Thin-C uses the pre-processor to help us.

Use the pre-processor to assist in type safety

A strong type has a name and an underlying type, a mutable strong type has a name and the same underlying type; so we could write two macros that take two parameters each (name and underlying type) but we need to ensure that

- the underlying type is the same for both macros
- the names only differ by the `Sink` suffix

The way to enforce this is to declare the underlying type separately and only have one parameter for each macro and that parameter is the name of the strong type (without the suffix), thus :

```
// simple token concatenation
#define TOKEN_CONCAT(A,B) A##B

// underlying type
#define UNDERLYING_TYPE(T) TOKEN_CONCAT(t,T)
#define DECLARE_UNDERLYING_TYPE(BASE,T) \
  typedef BASE UNDERLYING_TYPE(T)

// declaration of a strong type
#define _HIDDEN_(B) TOKEN_CONCAT(_hidden_,B)
#define HIDDEN_NAME _HIDDEN_(__LINE__)
#define DECLARE_STRONG_TYPE(T) \
  typedef struct { UNDERLYING_TYPE(T) const HIDDEN_NAME; } T

// mutable types
#define QK(T) TOKEN_CONCAT(T,Sink)
#define DECL_MUTABLE(T) \
  typedef struct { UNDERLYING_TYPE(T) HIDDEN_NAME; } QK(T)
```

Then we declare the underlying type, strong type and mutable strong type thus:

```
// declare the underlying type
DECLARE_UNDERLYING_TYPE(int, Hour);

// declare the strong type
DECLARE_STRONG_TYPE(Hour);

// declare mutable version of the strong type
DECL_MUTABLE(Hour);
```

This will generate three type declarations:

```
typedef int tHour;
typedef struct { tHour const _hidden_24; } Hour;
typedef struct { tHour _hidden_27; } HourSink;
```

We can actually ignore `tHour` since we won’t be using it in our programs but it was necessary to declare it to ensure that both strong types used the same underlying type. Notice that the member names are auto-generated. This is to prevent programmers from “unwrapping” strong types and using or manipulating the member variable directly thus violating their integrity. In order to use strong types in programs we need to use functions; and it assists greatly if these functions are named in a consistent way.

Use consistent naming

It doesn't matter what naming convention you use as long as you are consistent. That way, programmers can identify what is a type, what is a variable, what is a function, etc. The examples in this article use the following convention:

- types begin with an Uppercase letter and are in CamelCase. Underscores may be used to aid readability:
  `Hour, Minute, Base64Char, ElapsedTime, XML_Tag`
  (the exception is the underlying type — this is so that the underlying type won't clash with an actual strong type)
- variables are in lowercase:
  `self, result, status, hour, minute, elapsedtime`
- standard functions that manipulate strong types have a standard prefix in lowercase followed by the type name:
  `mkHour, vHour, dmHour`
- other functions use lowercase and underscores except where a type appears in the function name:
  `decode_upto_3, hhmm_to_Minute, elapsed_time`

Use the pre-processor to declare and define standard functions

Since the programmer can't directly access the member variable of a strong type (because it has an auto-generated name) we need to auto-generate a function to get the value of the member variable.

```
// declaration of a function to get the value of a strong type
#define VALUEOF(T) TOKEN_CONCAT(v,T)
#define DECL_VALUEOF2(T,P) \
  UNDERLYING_TYPE(T) VALUEOF(T)( T P )
#define DECLARE_FN_RETURNING_VALUEOF(T) DECL_VALUEOF2(T,)

// definition of a function to get the value of a strong type
#define DEFINE_FN_RETURNING_VALUEOF(T) DECL_VALUEOF2(T,self) \
  { return *(UNDERLYING_TYPE(T) const *)&self; }
```

Then in a header file we can declare the function thus:

```
// declare function to get Hour value
DECLARE_FN_RETURNING_VALUEOF(Hour);
```

And in a C file we can define it thus:

```
// define function to get Hour value
DEFINE_FN_RETURNING_VALUEOF(Hour)
```

Notice that there is no trailing semi-colon. This can confuse editors that use auto-indent, but there isn't a lot we can do about that.

In real code we can use the function just like any other function:

```
    if (vHour(hour) < 12) { /* do something */ }
```

Although it is tempting to extract the value into an `int` then just use the `int`, programmers must resist this — and it usually a sign that the program shape is wrong anyway; which is a topic we will revisit later.

"Solidify" mutable variables when mutability is not required

The `vHour` function can only take an argument of type `Hour`, C won't let you pass an `HourSink` argument to it. We could write a pre-processor macro to generate `vHourSink` but we don't require mutability in order to get a value (only when we set a value) so an alternative is to convert the `HourSink` variable to `Hour`. As you might expect, we can use a macro to generate a function to do this.

```
// "solidify" (de-mute) a value
#define DEMUTE(T) TOKEN_CONCAT(dm,T)
#define DECL_DEMUTE2(T,P) T DEMUTE(T)( QK(T) P )
#define DECLARE_DEMUTE(T) DECL_DEMUTE2(T,)
#define DEFINE_DEMUTE(T) \
  DECL_DEMUTE2(T,self) { return *(T*)(&self); }
```

Declare the function in a header file, define it in a C file and use it in your code:

```
// in header file: declare solidify function
DECLARE_DEMUTE(Hour);

// in C file: define solidify function
DEFINE_DEMUTE(Hour)

// example usage
   HourSink hoursink = { 3 };
   if ( vHour( dmHour(hoursink) ) < 12 ) { /* something */ }
```

Use inline

The `vType` and `dmType` functions are ideal candidates for inlining since all they do is take an address, cast to a pointer type, and dereference the address. To define one of these functions as inline we simply prepend the `inline` keyword as normal.

```
inline DEFINE_DEMUTE(Hour)
```

Don't forget that in C99 you also have to provide an external declaration:

```
extern DECLARE_DEMUTE(Hour);
```

Ensure variables are always in range

Very often, types represent information that has a limited range of values so our programs must enforce this range limit. This is more easily achieved if there is a single function that takes care of the range checking. For strong const types this is done using a "maker" function because once the value is set it won't change, so the value only needs checking once.

```
/* minutes-past-the hour
** range: 0-59
** we are not concerned about saving space,
**  so use natural int size
*/
DECLARE_UNDERLYING_TYPE(unsigned int, MinutesPast);
DECLARE_STRONG_TYPE(MinutesPast);

// declare the maker function for MinutesPast
MinutesPast mkMinutesPast( UNDERLYING_TYPE(MinutesPast) );

// define the maker function for MinutesPast
MinutesPast mkMinutesPast( UNDERLYING_TYPE(MinutesPast) a )
{
    MinutesPast rv = { a%60 };
    return rv;
}

// example usage
void to_or_past(unsigned int mm)
{
    MinutesPast mins = mkMinutesPast( mm );
    if ( vMinutesPast(mins) == 0 ) {
       // on the hour
    }
    else if ( vMinutesPast(mins) <= 30 ) {
        // vMinutesPast(mins) past
    }
    else {
        // (60-vMinutesPast(mins)) to
    }
}
```

The `mkMinutesPast` function silently converts the argument to a value that was in range. There are times, however, when we need a maker function to abort when given a value out of range.

```
#include <assert.h>

B41dChar mkB41dChar(char value)
{
    assert(isValidB41Char(value));
    B41dChar rval = {value};
    return rval;
}
```

For finer-grained assertions, see the section on *Design by contract* below.

Encapsulate cohesive variables

If two, or more, variables work together; that is, in technical terms they have high cohesion; then these variables should be encapsulated in a type. One of the common idioms in C is a pointer or an array with an associated length or offset — this is exactly the sort of situation that needs an encapsulated type.

```
typedef unsigned ByteCount;
typedef struct
{
    uint8_t *bytes;
    ByteCount nbytes;
} ByteSinkSeq;
```

The type encapsulates a sequence of mutable bytes (C type `uint8_t`) with a counter. Access to variables of this type is controlled by functions.

```
ByteSinkSeq mkByteSinkSeq( uint8_t *, unsigned );

typedef bool hasAdded;
hasAdded addByteToSink( ByteSinkSeq *, uint8_t );

hasAdded addAllBytesToSink(
    ByteSinkSeq *,
    uint8_t const *bytes,
    ByteCount
);
```

There are a few points worth mentioning here. Firstly, the member variables of the `struct` in this example are not strong types. We could have made them strong, or some of them strong, or none of them. Secondly, the member variables do not have hidden names — this would be too difficult to implement safely. There are ways of hiding struct membership by only declaring a forward reference in the header file then using dynamic allocation to instantiate instances of the type, but this brings with it its own problems. Thin-C doesn't recommend any of these data hiding techniques, but nor does it discourage them. Thirdly, the publicly-accessible interface uses separated function parameters (pointer to bytes and a count) rather than an encapsulated type — the decision as to which to use would be driven by the most common use case.

In the next example we define a type that is used to hold the result of decoding a triplet of characters from a 41-letter alphabet into a 16-bit quantity.

```
// Result of decoding a Triplet of Base41 Chars

typedef uint8_t CountTo2;
typedef struct
{
    B41D_Status status;
    uint16_t value;
    CountTo2 count; // number of bytes decoded to
                    // 2: 3 chars have decoded to 2 bytes
                    // 1: 1 or 2 chars have decoded to 1 byte
                    // 0: no chars decoded
} B41D_Result;

B41D_Result initB41D_Result(void);

B41D_Result decode( UpToThreeB41DChars const * );

hasAdded xfer_to_sink(B41D_Result const *, ByteSinkSeq *);
```

The type and the interface are quite simple. The `initB41D_Result` function sets all members to initial values. This is like a `mk` function, but takes no parameters. The `decode` function decodes up to three characters and results in a `B41D_Result` type. The only other function in the interface is `xfer_to_sink` which writes the bytes of the resulting value to the byte sink. The `B41D_Status` type (and hence the `status` variable) indicates whether the `decode` was successful or whether there was overflow or underflow. The `value` member holds the resulting bits and the `count` member indicates how many *bytes* (and thus how many bits) the `value` holds. However, if there was overflow or underflow then there is uncertainty about the meaning of the `value` and `count` members. Also, if zero characters were decoded (because the `UpToThreeB41DChars` argument contained zero characters) then what does the `value` member hold? There are two schools of thought on this issue: one school recommends that in such cases member variables should be set to a well-known value (*e.g.* an initial value or a trap-representation) the other school says that the member variables' values are *undefined*. Thin-C says "never assume" (one school or the other).

Never assume correctness

This is a really tough mindset to get into; and it is a mindset. It can be assisted by programming techniques, but at the end of the day, it's the mind that decides. All the time you have to be thinking that everything you have ever written is incorrect, and usually incorrect in the most un-obvious way (but sometimes they *are* just plain wrong). The gremlins are in charge. Murphy's Law always applies. Someone is right now building a better idiot. So what does a paranoid programmer do to root out incorrectness? The answer has three parts:

1. design for verification
2. verify in code
3. verification testing

Since Thin-C is mostly about *writing* code we will look at point 2 first…

Verify in code

Before you do anything, check that it is alright to do so — and remember to never assume that the caller or the callee will do the checking for you. This is all part of defensive programming.

Do not be afraid to verify.

For example, using the `xfer_to_sink` function above. The function should check that the status is OK before trying to add to the Sink. And the caller should check this also. The one should not rely on the other — otherwise a tighter coupling than necessary is created.

Design by contract

A form of verification-in-code is sometimes known as *design by contract* which uses pre- and post- conditions, invariants and assertions. This is not a substitute for look-before-you-leap checking and input cleansing but a way of detecting run-time data value and data relationship anomalies. If implemented in the right way, various levels of checking can be turned on or off at compile time thus allowing greater assertion checking during the development phase and increased performance (but less checking) for released code. An example header file is given in Appendix A.

Design for verification

The best way to improve verification is to design it in from the start. Firstly, try to make it easy to detect success. Usually this happens anyway, and by accident. But we don't like things happening by accident, so make a conscious effort to ensure that test-for-success is easy. Secondly, we want to design failure cases in such a way that we can determine that the item under test failed in the expected fashion. It is not good enough to return a boolean to indicate that "an error occurred" we need to capture *what* has failed and *why* it has failed. But before we decide how we are going to do that we need to make a *complete list of all* possible failure modes. Then we have to ensure that we handle every one of them. And in each error handler provide enough information to the caller to determine where the error occurred. C programmers have tended to shy away from this amount of feedback, relying instead on error codes and logging to stderr; but having comprehensive feedback assists greatly in verification and could be a real boon to those using your code.

Test everything

It is one thing to write checks and verifications into your code, it is another thing to exercise those checks thoroughly. Unit tests are the means to perform those exercises. And as your gym instructor will tell you, the more you exercise the fitter you become. If you want your software to be fit (for purpose) you have to exercise it. And that doesn't mean running the same test 100 times; but writing 100 different tests and running them every time the code changes.
Unit tests are great. But if your units are big, with lots of code paths, then your unit testing will become less effective, because of the difficulty in exercising all the code paths in proportion to their importance or criticality.

Write small functions

The smallest testable unit in C is the function. So keep your functions small and simple so that they will be easier to test. The general rule is that a function should do one thing and do it well. The easiest functions to test are those that have no inputs and only one output (e.g an initialisation function) you only have to write one test for these. The next easiest are those that have one simple input and one output derived directly from the input. As function complexity increases, testing becomes more challenging. There are several factors that increase function complexity such as the number of parameters, steps, branches, etc.
As the complexity of the parameters, the number of parameters and the number of branches increases the number of tests one has to write increases exponentially. However, as the number of *steps* increases it is testability itself that becomes a challenge. How can we be certain that every step has behaved correctly? Just because the black-box tests all pass doesn't mean that the input to one step, for example, isn't masking an error in the output of a previous step. It would be good if we could somehow isolate each step in a function and test its inputs and outputs in the same way that one might be able to do so with an electronic component on a circuit board, for example. The closest we can come to that, without writing an auto-debugger, is to make every step in a multi-step function a call to a sub-function; the sub-functions can then have unit tests written for them. We can also write unit tests for pairs of sub-functions, for example.

Shun functions with multiple parameters of the same type

We can't say "*avoid* functions with multiple parameters of the same type" because there are many perfectly valid cases for having such functions (such as adding or subtracting two quantities together) so *shun* was chosen because it is the shortest synonym of *avoid* we could find in the thesaurus.

If a function has two or more parameters of the same basic type it may be an indication that some typedef'ing is required. Take one of the remote command functions defined in `unistd.h` as an extreme example:

```
int rcmd_af(char **, int, const char *,
         const char *, const char *, int *, int);
```

Two of the string parameters are actually of the same logical type (they are user names) the other, however, is a command string. The pointer-to-string points to a hostname. One of the integers is a port number and the other is an address family. The pointer-to-integer is the address of a file descriptor. Since we can't change a standard function we could write a simple wrapper:

```
SocketDescriptor remote_cmd_addr_fam(
    HostName *,
    PortNumber,
    UserName locuser,
    UserName remuser,
    FileDescriptor *,
    AddressFamily
);
```

Our code, if written properly, would be using these types anyway.

Avoid boolean parameters

This time we can say *avoid*. A boolean parameter is usually a sign that a function is two flavours in one. It also tells you nothing about its intended usage. Compare these two function prototypes that include the parameter names:

```
Status write_xml_end_tag(FileDesc fd, bool keepopen);
Status write_html_end_tag(FileDesc fd, bool closeonreturn);
```

Both functions have the ability to close the file descriptor when they have written the end tag; but one author decided that `true` meant "keep the file open" and the other decided that `true` meant "close the file". One improvement would be to define a type that better reflected the action:

```
typedef enum { keepopen, closeonreturn } WriteEndTagAction;

Status write_xml_end_tag(FileDesc, WriteEndTagAction);
Status write_html_end_tag(FileDesc, WriteEndTagAction);
```

A better improvement would be to define four functions:

```
Status write_xml_end_tag(FileDesc fd);
Status write_xml_end_tag_and_close(FileDesc fd);
Status write_html_end_tag(FileDesc fd);
Status write_html_end_tag_and_close(FileDesc fd);
```

The non-closing versions of the functions would be simpler than they were previously and the closing versions would simply call the non-closing version and then close the file descriptor.

Show your working out

Expressions that contain several terms and different operators are:

- more difficult to read
- more difficult to debug
- more difficult to be defensive with

For example, this expression is used in the calculate of a CRC:

```
rv=crcTable[*msg++^rv&255]^rv>>8;
```

If we wanted to evaluate the expression in the square brackets, say, to find out whether that expression went out of the array bounds, then it would be a bit painful even in a half-decent debugger. We can't defend against an out-of-bounds read unless we repeat the expression (omitting the post-increment) in a prior assertion statement. And the expression is just plain difficult to read.
One advance would be to *write small functions*; then the expression becomes:

```
rv = xor(lookup(xorb(*msg++,lo8bits(rv))),rshift8(rv));
```

Here, at least, we could augment the functions with assert checks or could put breakpoints in the functions if we are stepping through with a debugger. But the expression is still difficult to read. A little bit of whitespace and some more meaningful variable names can help. We also separate the post-increment from the expression (not shown)

```
        remainder = xor(
            lookup(
                xorb(
                    *message,
                    lo8bits(remainder)
                )
            ),
            rshift8(remainder)
        );
```

This is better, but it is still less intuitive than it can be; and the return types of each of the sub-functions aren't explicit. We need to be clear about each step in the calculation. In other words: *show your working-out*.

```
        Byte loByte = lo8bits(remainder);
        Byte key = xorb( *message, loByte );
        crc_t crcTableValue = lookup(key);
        crc_t shifted = rshift8(remainder);
        remainder = xor(crcTableValue, shifted);
```

or, if you prefer to see the C binary operators in action:

```
        Byte loByte = lo8bits(remainder);
        Byte key = *message ^ loByte;
        crc_t crcTableValue = lookup(key);
        remainder >>= 8;
        remainder ^= crcTableValue;
```

Use less of C

Thin-C attempts to keep things simple in order to reduce errors. One of the ways in which it achieves simplicity is to limit itself to only the essential elements of C. Therefore, Thin-C doesn't use `goto`, `continue` or `do-while`. It only uses `break` from within a `switch` statement. It doesn't use `auto`, `register` or `restrict`. Furthermore, to keep access to structs as consistent as possible, Thin-C always tries to use the `->` operator in preference to the `.` operator. This means that when it declares a local variable of some struct type it will declare it as an array of size 1. (This rule doesn't apply to "hidden" structs such as strong types)

```
B41D_Result result[1] = {initB41D_Result()};
```

This way of declaring struct variables assist greatly in refactoring prototype code. The corollary to this little practice is there is less need to use the & (address of) operator when passing arguments to functions that take pointers.

Prevent "design leap"

Design Leap is when the programmer jumps directly from a requirement to code without actually thinking about the data that is being processed. For relatively simple processing tasks most experienced programmers are so good at this that they can even get their unit tests to pass first time most of the time. However, the code will most likely have a very awkward "shape" and might be difficult to re-factor when the time comes to do so.

Let's take a simple example, given a string that holds a sign-less real number, find the number of significant digits in that string (that is, the number of digits inclusive between the first non-zero digit and the last non-zero digit). Perhaps you should try to write this program before continuing reading this article.

After a few development iterations (to cope with errors and zeroes) I came up with the following:

Firstly, a struct to hold the result of scanning the string for the first and last non-zero digits, the decimal point and the terminating character (should point to a NUL unless the input contained invalid characters):

```
typedef struct
{
    char const *firstnz; // first non-zero
    char const *lastnz;  // last non-zero
    char const *dp;      // decimal point
    char const *end;     // terminating char
} SigDigs;
```

The function that does the scanning:

```
SigDigs find(char const *ptr)
{
    SigDigs rv[1] = {{NULL,NULL,NULL,NULL}};
    for (;
        isdigit(*ptr) || ((*ptr == '.') && (rv->dp == NULL));
        ptr++
    )
    {
        if (*ptr == '.')
            rv->dp = ptr;
        // it's a digit
        else if (*ptr != '0')
        {
            rv->lastnz = ptr;
            if (rv->firstnz == NULL)
                rv->firstnz = rv->lastnz;
        }
        // otherwise it is zero - which we can ignore
    }
    assert( IMPLIES((rv->lastnz!=NULL),(rv->firstnz!=NULL)) );
    rv->end = ptr;
    return rv[0];
}
```

(The function that calculates the number of significant digits from the result is straight-forward and is not included here.)

Although the `find` function is fairly short and gives the correct results it suffers from a particularly prevalent anti-pattern: the "robial" (run-once branch in a loop). This is where in the body of a loop there is an if-statement that is only ever taken once but doesn't cause the loop to terminate. Sometimes this pattern is obvious:

```
bool firsttime=true;
while (condition)
{
    if (firsttime) { do-something; firsttime=false; }
    else { do-something-else; }
}
```

But much of the time it is less obvious. In our `find` function we have:

```
        if (*ptr == '.')
            rv->dp = ptr;
```

This branch can only ever be run once in this loop because the loop invariant contains the clause:

```
        (*ptr == '.') && (rv->dp == NULL)
```

Which prevents us entering the loop if we encounter a full stop when the `dp` member variable is already set (i.e. we have already seen a full stop).
Despite its compactness, and its neatness, the loop hides the shape of the input data, which is actually a form of obfuscation.

Match the shape of an operation with the shape of the data.

To prevent obfuscation, Thin-C recommends that the shape of an operation matches the shape of the data it is operating upon. This forces the programmer to consider (and hopefully document!) what shape the input data has and can prevent design leap. In the significant-digits exercise above, the shape of the input is as follows:

```
many: zero-digit '0'
optionally-one-of:
  dot-first
  digit-first
```

This states that the input consists of many (that is, zero or more) `'0'`s followed by either the "dot-first" shape, the "digit-first" shape or nothing (the end of the input). (We have omitted the fourth option (that it encounters an error) purely for this example. We would normally enumerate the error cases.)

The dot-first shape is when the input contains a decimal point before any non-zero digits:

```
dot-first =
  decimal-point ‘.’
  many: zero-digit ’0’
  optional:
    first-non-zero-digit [1-9]
    many: digit [0-9]
```

The digit-first shape is when the input contains a non-zero digit before the decimal point (or doesn’t contain a decimal point at all):

```
digit-first =
  first-non-zero-digit [1-9]
  many: digit [0-9]
  optional:
    decimal-point ‘.’
    many: digit [0-9]
```

In both shapes it is only possible to encounter a decimal point once; similarly it is only possible to encounter a `first-non-zero-digit` once. Did you spot that in the `find` function there was a second *robial*?

```
            if (rv->firstnz == NULL)
                rv->firstnz = rv->lastnz;
```

The resulting code from this design is much longer than the `find` function given above; but we can be more confident that it is correct. Firstly, we can review the design presented above and be confident that it captures the shape of the input. If we tried to reverse-design the `find` function then we would struggle to match this design to the shape of the input and we would be less confident that it is correct. If we are confident in the design we can compare the code to the design and be confident that the code is also correct.

Keep loops simple

One way to avoid *robials* is to make your loops no more complicated than this:

```
    for (i=0; (i<n) && side_effect_ok(obj, args[i]); i++)
        array[i] = obj->value;
```

The body of the loop should contain only simple actions — ones that do not require guards or pre-conditions. Ideally there should only be one action, such as an assignment or a call to a void function. The loop invariant should have no more than two clauses; the first clause is the simple array-bounds check or equivalent and the

second clause is a boolean function with side effects that performs the input conversion and error checking. The function should populate a struct with information about errors detected and results of input conversion.
The code snippet above will actually break the coding standards of a lot of organisations. Firstly, they insist on curly braces for every loop and if-statement. In Thin-C we simplify the code so much that nearly every compound statement has only one expression statement. Braces would just be unnecessary furniture.
Secondly, they insist that loop conditions have no side effect. Again, the norm in Thin-C is for loops to do all their work in the function that is called in the condition. This is what Thin-C programmers *expect* so there can't be any surprises from the side-effects (which is the main reason why the writers of such coding standards insist on avoiding functions with side effects — they are just not used to them). One of the benefits of the loop shape given in the example is that the third clause of the `for`-loop is not executed if the function returns `false`; this leaves the loop counter referring to the array member that caused the error. If the function was moved to the body of the loop then we would have to use a `break` statement to achieve the same result.
Let's take an example program. This program takes a sequence of arguments that should be strings representing signed long integers. The program checks that each argument is a valid integer and also that the same integer does not appear more than once in the sequence of arguments. Here is an outline of the program's `main`:

```
int main(int argc, char*argv[])
{
    long *array = NULL;
    unsigned nargs = (unsigned)(argc-1);
    if (nargs == 0)
        fprintf(stderr, "no arguments!\n");
    else if (NULL == (array = malloc(nargs*sizeof(*array))))
        fprintf(stderr, "malloc error!\n");
    else if (!are_args_ok(argv+1, nargs, array))
        fprintf(stderr, "invalid arguments\n");
    else
        for(int i=0; i<(argc-1); i++)
            printf("[%s] => %ld\n", argv[i+1], array[i]);
    // clean up
    if (NULL != array)
        free(array);
    return 0;
}
```

In order to detect duplicate arguments, the program `malloc`s space for an array of `long`s. If there are no arguments, there is nothing to `malloc` so the program will print a message in this case. All of the work of the program is performed by the `are_args_ok` function. If everything is successful the program prints out the arguments with their corresponding integers. The shape of this function is the typical shape of functions in Thin-C and is sometimes called the "chain of dependencies" — where each step depends on the previous step — but instead of having a collection of nested `if`-statements we have this elegant line of `if`s.

Keep error actions close to error checks

Thin-C avoids excessive nesting by always checking for error instead of checking for success and by keeping the error action next to the error check. A chain of dependencies is therefore constructed from a repeated building block:

```
if ( failed == action(args) )
    error_action();
else
```

The action is usually a function with side-effects and quite often is an assignment expression (another thing that breaks coding standards! again, because the writers of these standards are just not used to seeing assignments in conditions). In the `main` above we have:

```
else if (NULL == (array = malloc(nargs*sizeof(*array))))
    fprintf(stderr, "malloc error!\n");
```

There is a function with side-effects (`malloc`) and an assignment (`array =`) but there is still an explicit check against `NULL`. There is absolutely nothing wrong with this statement. It is perfectly readable and keeps the flow of the function simple.

Continuing with our program, the `are_args_ok` function referred to in `main` has the standard Thin-C loop shape:

```
static bool are_args_ok(
    char *args[],
    unsigned narg,
    long *array
)
{
    unsigned i;
    for(i=0; (i<narg) && is_number_ok(args[i], array, i); i++)
        ;
    return i==narg;
}
```

… with all the work performed in the `is_number_ok` function. This function is responsible for converting an argument to an integer and appending it to the array if the conversion was successful and the integer isn't a duplicate.

```
static bool is_number_ok(
    char const *str,
    long *array,
    unsigned nitems
)
{
    bool ok = true;
    char *endptr;
    errno = 0;
    long deco = strtol(str, &endptr, 10);
    if (errno != 0)
        ok = false;
    else if (*endptr != '\0')
        ok = false;
    else if (is_in_array(deco, array, nitems))
        ok = false;
    else
        array[nitems] = deco;
    return ok;
}
```

The final piece of the program (apart from any `#include` statements) is the `is_in_array` function.

```
static bool is_in_array(
    long key,
    long const *array,
    unsigned n
)
{
    unsigned i;
    for (i=0; (i<n) && (array[i]!=key); i++)
        ;
    return (i<n);
}
```

Again, the standard loop shape is in evidence although there is no need for a function with side effects here. Another benefit of the two-clause conditional is that at the end of the loop the first clause (the array-bounds check) will still be true if the second clause caused the loop to terminate and will be false if the loop got to the end of the array. We can always use this information to determine which clause caused the loop to end and thus whether there was an error.

Review and refactor

A Thin-C programmer is always dissatisfied with functions that don't conform to Thin-C standards and is constantly reviewing code to find conformance opportunities. The program we wrote above is a candidate for a little re-factoring.

Firstly, it doesn't use as much `typedef` as we would like. Secondly, it doesn't provide any useful feedback if it failed to convert an argument to an integer or there was a duplicate. Thirdly, the C standard function `strtol` just doesn't conform to Thin-C. We will tackle this third issue by defining a cohesive `struct` and by wrapping the function. From reading the manual we can see that `strtol` generates three pieces of information: the value of the integer, a pointer to the end of the converted characters and an error number; so we can encapsulate this:

```
typedef long int Integer;
typedef int ErrorNumber;
typedef struct
{
    Integer value;
    char *endptr;
    ErrorNumber errnum;
} StrTol;
```

the simplest wrapping function is:

```
void str_to_integer(StrTol *self, char const *str)
{
    errno = 0;
    self->value = strtol(str, &(self->endptr), 10);
    self->errnum = errno;
}
```

The function that converts a `StrTol` to an integer can have four possible outcomes:

- converted ok
- invalid argument (e.g. an invalid character, or only whitespace)
- overflow (too many digits to convert to a valid long integer)
- duplicate

So we define a type for this:

```
typedef enum { ok, outofrange, badarg, duplicate } Outcome;
```

and a type to hold this plus the value of the conversion:

```
typedef struct
{
    Outcome outcome;
    Integer value;
} ConvertedNumber;
```

The function to do the conversion and check whether it is a duplicate:

```
typedef unsigned int NItem;
typedef struct
{
    Integer const *a;
    NItem n;
} ArrayOfInteger;

void convert(
    ConvertedNumber *self,
    StrTol *st,
    ArrayOfInteger *array
)
{
    if (ERANGE == st->errnum)
        self->outcome = outofrange;
    else if (st->errnum != 0)
        self->outcome = badarg;
    else if ( st->endptr[0] != '\0' )
        self->outcome = badarg;
    else if (is_in_array(st->value, array))
        self->outcome = duplicate;
    else
        self->outcome = ok;
    self->value = st->value;
}
```

(We have also modified the `is_in_array` function to take an `ArrayOfInteger` parameter — this is not shown here.)
The higher-level function that combines the two steps:

```
bool convertedOK(
    ConvertedNumber *self,
    char const *str,
    Integer const *array,
    NItem nitems
)
{
    StrTol st[1];
    // str => StrTol
    str_to_integer(st,str);
    // StrTol + Array => ConvertedNumber
    ArrayOfInteger ai[1] = {{array, nitems}};
    convert(self, st, ai);
    return (self->outcome == ok);
}
```

We are getting closer to the top level now. To assist in unit testing the program needs to know which argument caused the error and what the error was.

```
typedef struct
{
    Outcome outcome;
    char const *badone;
} ArgsOk;
```

So the `are_args_ok` function now takes this struct as a parameter.

```
bool are_args_ok(
    ArgsOk *self,
    char *args[],
    NItem narg,
    Integer *array
)
{
    NItem i;
    ConvertedNumber cn[1] = {{ok,0}};
    for (i=0;
        (i<narg) && convertedOK(cn, args[i], array, i);
        i++
    )
        array[i] = cn->value;
    self->outcome = cn->outcome;
    self->badone = (cn->outcome == ok) ? NULL : args[i];
    return (i==narg);
}
```

One additional difference to the previous version of this function is that the statement that appends the converted integer to the array is now in the body of the loop rather than being in the function-with-side-effect. The `main` function now has the ability to print out the reason for any error and the argument that caused it. This is left as an exercise for the reader.

Use declarative statement comments

With the extensive use of types and small functions comes the ability to express operations using simple (pseudo-) syntax. Programming then becomes closer to chemistry or algebra. The example program above use five types: argument strings, `ArrayOfInteger`, `StrTol`, `ConvertedNumber`, and `ArgsOk`. The allowable conversions between these types:

```
    argument-string => StrTol
    StrTol + ArrayOfInteger => ConvertedNumber
    ConvertedNumber + ArrayOfInteger => ArrayOfInteger'
    ConvertedNumber => ArgsOk
```

With these conversions we can see that it is possible using algebraic substitution to derive other conversions:

```
    argument-string + ArrayOfInteger => ConvertedNumber
    argument-string + ArrayOfInteger => ArgsOk
    argument-string + ArrayOfInteger => Array'
    many:argument-string + ArrayOfInteger => ArrayOfInteger' +
ArgsOk
```

The last statement summarises the whole program. The program arguments are converted to an array of `Integer` plus information about success/failure.
These sorts of statements are ideal for program comments since they are very succinct yet carry a high degree of information.

Use indentation as a measure of hierarchy

You may have noticed that the examples in this article all use a particular style of indentation and positioning of the closing brackets. This is another feature of Thin-C and it contributes greatly to readability of code. The first rule is that indentation is a measure of hierarchy. If something is sub-ordinate to something else (and doesn't fit on one line) then all the sub-ordinates are *always* indented by four spaces. This means that just by looking at the indentation you can determine where something fits in the hierarchy. Let's take the fairly extreme example from the section on *show your working out*:

```
        remainder = xor(
            lookup(
                xorb(
                    *message,
                    lo8bits(remainder)
                )
            ),
            rshift8(remainder)
        );
```

if we remove the parentheses and commas we can still immediately see the hierarchy

```
        remainder = xor
            lookup
                xorb
                    *message
                    lo8bits remainder
            rshift8 remainder
```

Notice how `lookup` and `rshift8` are obviously arguments to the same function even though they are several lines apart. This is because of the clarity of the indentation.
The second rule is that the closing bracket must be on a line with the same amount of

indentation as the opening bracket. If the closing bracket is on the same line as the opening bracket then obviously the closing bracket is on a line with the same amount of indentation as the opening bracket. If the expression is split across lines then the closing bracket is the first non-whitespace character on a line. This means that the closing bracket is never stuck on the end of a sub-ordinate line; because the bracket does not belong to the sub-ordinate but to the level above. If we removed all but the brackets and commas from the example:

```
(
    (
        (

            ,
            ()
        )
    ),
    ()
);
```

we can clearly see which bracket belongs to what.

Auto-generate if possible

One of the goals of Thin-C is to make programming so simple that a computer could do it. One area where auto-generation is possible is in the creation of the standard functions for strong types, but it would be nice if there was a formal higher-level notation that we could use to specify types and operations on types. However, there is so little redundancy in a Thin-C program that any higher-level notation that we could invent in which we could specify a program would only be slightly shorter than the C program generated from it. The two areas where there is some redundancy are function parameters and function names.

Because Thin-C makes heavy use of `typedef`, most functions have a set of parameters all of which are of a different type. This would mean that both the parameter type and the parameter name would be unique within the function, so we could dispense with the parameter name if we were specifying the function in some higher-level notation. Another effect of heavy `typedef` use is that most function signatures are unique within a program — that is, each function has a unique combination of parameter types — so we could dispense with the function names when writing the program specification. Other than that, the only savings we could make would be to dispense with some brackets by using indentation instead.

One area of auto-generation worth exploring is the derivation of functions using the "algebraic substitution" hinted at in the *Use declarative statement comments* section; given a set of function signatures such as `A=>B A=>C B+C=>D` it should be possible to generate a function with a signature `A=>D`.

Appendix A — assertion.h

```
#ifndef ASSERTION_H
#define ASSERTION_H

// compile your program with, for example:
// cc -DASSERTION_LEVEL=ASSERTION_ALL

#include <assert.h>

#ifndef ASSERTION_LEVEL
#define ASSERTION_LEVEL 0
#endif

#define ASSERTION_ONLY 1
#define ASSERTION_LEVEL_REQUIRE 2
#define ASSERTION_LEVEL_ENSURE 3
#define ASSERTION_LEVEL_INVARIANT 4
#define ASSERTION_ALL 5

#if ASSERTION_LEVEL >= ASSERTION_ONLY
#define ASSERT(E) assert(E)
#else
#define ASSERT(E) ((void)0)
#endif

#if ASSERTION_LEVEL >= ASSERTION_LEVEL_REQUIRE
#define REQUIRE(E) assert(E)
#else
#define REQUIRE(E) ((void)0)
#endif

#if ASSERTION_LEVEL >= ASSERTION_LEVEL_ENSURE
#define ENSURE(E) assert(E)
#else
#define ENSURE(E) ((void)0)
#endif

#if ASSERTION_LEVEL >= ASSERTION_LEVEL_INVARIANT
#define INVARIANT(E) assert(E)
#else
#define INVARIANT(E) ((void)0)
#endif

#if ASSERTION_LEVEL >= ASSERTION_ALL
#define CHECK(E) assert(E)
#define WHERE(E) assert(E)
#else
#define CHECK(E) ((void)0)
#define WHERE(E) ((void)0)
#endif

#define IMPLIES(P,Q) (!(P) || (Q))
#define IFF(P,Q) (!(P) == !(Q))
#define XOR(P,Q) (!(P) != !(Q))

#endif
```