Robust Design Techniques for C Programs

David Turner
(david@freetype.org)

Abstract

This paper presents several techniques used to design robust C programs and libraries.

Table of Contents

Introduction

This paper contains a survey of four major techniques used to design robust code in C. It tries to highlight the strengths and annoyances of each one of them in the eye of the typical C developer. This mainly means in terms of ease-of-use (both in writing and maintaining the source) and runtime cost (does it make the program code bigger, or slower ?).

It also introduces a lesser-known technique, called "*cleanup stack exception handling*" (CSEH), and shows that it is the only technique that isn't delicate or painful to use from a developer's point of view.

Finally, it goes further by analysing CSEH, by providing portability and performance estimates, requirements as well as list a few real-world examples.

I. Robust Design Techniques

It's important to understand that robust programs generally need to deal with three kinds of "exceptional" conditions:

- **user errors**, when invalid input is passed to the program.
- **exhaustions**, when the program tries to acquire shared resources.
- **internal errors**, due to bugs (e.g. dangling pointers).

A reliable program must ideally detect or prevent all of these conditions, and deal with them in a *safe* and *intelligent* way.

Here, the term "safe" means that the program won't crash, won't start barfing garbage at the console or window, and won't try to do nasty things like deleting files or creating 64 levels of nested directories, etc.. In other words, it will keep the current system in a stable state.

The term "intelligent" is more difficult to detail. A reasonnable assumption is that it should try to provide feedback in case of user errors, and always recover gracefully from the exceptional condition once detected. This means that the program is supposed to *keep running normally*, even if it's with reduced capabilities.

Finally, note that certain bugs, like memory leaks, are not treated as exceptional conditions and will not be detailed or handled by the techniques presented in this paper. A reliable C program should however ideally prevent them too.

Applied to the C programming language, the first steps towards reliability are simply to perform a great number of *runtime checks*. There are several well known techniques to do that, of which a few are:

- **checking assertions**, to detect un-respected constraints
- **checking array indexing** whenever possible.
- **embedding magic numbers** within objects, to check their type quickly.
- **checking the result** of any system call or library function.

These steps are important to *detect* exceptional conditions, but the real hard work is in determining *how* to deal with them in a way that is both "safe" and "intelligent". We will now present several techniques used by C developers; later, we'll compare their relative advantages.

I.1. Being *extremely* paranoid:

The most common technique used by C developers to deal with exceptional conditions is simply to *check everything everywhere*. Basically, this consists in assuming that nearly every function call *might* fail for some reason. Whenever failure is detected, the program must perform "cleanup" to "undo" the current operation.

As a consequence, 90% of the functions written using this convention are *designed to return an error code*, and nearly all *function calls are checked for errors*. The code also needs to ***explicitely*** destroy or release the objects or resources that were created/acquired before the error.

As a simple example, the following "unsafe" code doesn't check memory exhaustions:

```
void  do_something( .... )
{
  char  *block1, *block2, *block3;

  block1 = (char*) malloc( 1024 );
  block2 = (char*) malloc( 1024 );
  block3 = (char*) malloc( 1024 );

  .... // do something

  free( block3 );
  free( block2 );
  free( block1 );
}
```

But it can be converted into the following to make it reliable with the paranoid technique:

```
int   do_something( .... )
```

```
{
  char  *block1, *block2, *block3;

  block1 = (char*) malloc( 1024 );
  if ( block1 != NULL ) {
      return ERR_MALLOC;
  }

  block2 = (char*) malloc( 1024 );
  if ( block2 != NULL ) {
      free( block1 );
      return ERR_MALLOC;
  }

  block3 = (char*) malloc( 1024 );
  if ( block3 != NULL ) {
      free( block2 );
      free( block1 );
      return ERR_MALLOC;
  }

  .... // do something

  free( block3 );
  free( block2 );
  free( block1 );

  return ERR_OK;
}
```

Note how we had to change the function's signature (to return an error code), and the special code that was introduced to deal with the potential exhaustions.

The paranoid scheme is simple and straightforward to implement and it is widely used for system calls and libraries. However, it has a number of important inconveniences too:

- it is extremely fragile, because "forgetting" to check a function call may result in critical bugs (like resource leaks) that may be extremely difficult to detect in the future, since they would only occur in very, very, specific conditions. It's also very hard to check the code that handles the exceptions, since it is never executed in typical runs of the program.
- it adds significant size to the source code and can also be a potential performance problem (adding many jumps to your code for the sake of exception handling decreases compiler optimization opportunities. Of course, that's more important when running on a 16 Mhz processor than on a 1 Ghz Athlon).
- it simply is *painful*. In the above example, the "unsafe" code is *considerably* faster to read, write and maintain than its "safe" variant.

Note that a simple but effective optimisation of the previous technique dramatically reduces the amount of "cleanup" code. It consists in *recording the operation's history* in order to later "undo" it *at a single place*. For example, the previous safe code fo our `do_something` example can be rewritten as:

```
int   do_something( .... )
{
  int   error;
  char  *block1, *block2, *block3;

  error  = ERR_OK;
  block1 = NULL;
  block2 = NULL;
  block3 = NULL;

  block1 = (char*) malloc( 1024 );
  if ( block1 == NULL ) goto Fail;

  block2 = (char*) malloc( 1024 );
  if ( block2 == NULL ) goto Fail;

  block3 = (char*) malloc( 1024 );
  if ( block3 == NULL ) goto Fail;

  .... // do something

Exit:
  if (block3) free( block3 );
  if (block2) free( block2 );
  if (block1) free( block1 );

  return error;

Fail:
  error = ERR_MALLOC;
  goto Exit;
}
```

In the above code, the temporaries are cleaned at a single location: the Exit label. The cleanup code's behaviour depends on the operation's history, which was recorded implicitely in the `block1`, `block2`, `block3` and `error` local variables. This dramatically reduces the number of redudant calls to `free` compared to the earlier version.

Note that this doesn't reduce the complexity of the safe code. It makes it clearer but still requires the programmer to take some heavy special measures to deal with exceptions reliably. Most notably, each function still needs to return an error code..

Many reliable C programs and libraries are implemented using a paranoid scheme. When one looks at the source code of large projects like XFree86, Ghostscript, FreeType, etc.., one will notice that their code is simply *littered* with checks..

This however isn't the norm of most developers, who often take deliberate measures to avoid dealing with annoying exceptions. For example, many, many, programs use a custom memory allocation function that *simply aborts* the program whenever a *memory exhaustion* condition is met.

This isn't surprising; since *memory allocation is so frequent*, choosing to ignore its failure reduces the number of error checks by more than 90% in a typical program. Actually, many C and C++ programming frameworks even *encourage* developers to assume such a behaviour since it makes life "so much easier".

That's probably OK for many cases (desktop applications, shell tools, etc..), where you can still tell your users to "buy more RAM", but is simply ***un-acceptable*** for system-level libraries, embedded systems or programs that need to be kept running wathever happens (like system services and daemons).

Fortunately, we'll see that it's possible to do much better than that..

I.2. Implementing *transactions*:

There is a second, less-known scheme, based on the "commit" and "rollback" concepts used in database systems. As we'll see, it's much more convenient than just being paranoid, but often lack the flexibility needed for large-scale development, which limits it to specific uses.

The master idea is to identify certain operations that we'll name "transactions", that need to be performed in a one-or-nothing fashion. This means that, in the event where the operation couldn't perform correctly (due to an exception), the program should be reset to the exact state it had before launching it.

Even though the "paranoid" scheme can also provide this guarantee easily, transactions implement it in a radically different way, with the help of the portable "`setjmp`" and "`longjmp`" instructions. (If you don't know them well, I'll encourage you to read **Annex A** of this document *now*).

For each transaction, a special object is designed to hold the history, or "state" of the operation's execution. This "state object" really contains:

- An history of all temporaries created, or resources acquired during the transaction.
- A jump buffer that is used to exit immediately from one of the transaction's function *as soon as* an exception is detected.

Here is some pseudo code that explains what happens during a transaction call:

```
state = new_transaction_state( ... );
if ( setjmp( state->jumpbuff ) == 0 )
{
  transaction_function_1( state, .... );
  transaction_function_2( state, .... );
}
else
{
  .. an error occured.. notify the caller, wathever ..
}
destroy_transaction_state( state );
```

This can be detailed as:

- First of all, a transaction-specific object must be created by the client program. It is initially empty.
- The program calls `setjmp` in order to record the current processor state in the transaction state's jump buffer.
- The `setjmp` call always returns 0 the first time it is called. We then call various functions that perform the actual transaction.
- Each function within the transaction updates the state according to the temporaries it creates and resources it acquires. As soon as an exception is detected within any of these functions, "`longjmp`" is called *immediately* to exit from the current execution context and transder control directly to the "`setjmp`", that will now return a non-0 value.
- If the `setjmp` returned a non-0 value, we do something in order to inform the program (usually setting an error flag or variable).
- Finally, we destroy the transaction state. The latter must be designed specially to ensure that the execution history is always rolled back in the exact opposite order or creations/acquisitions operated during the transaction. This should get rid of *all* temporaries created during the transaction.

This scheme's biggest benefit is the fact that *functions do not need to return error codes*, and function calls do not necessarily need to be checked for failure. Exception management and cleanup are handled "upstream", since the use of "longjmp/setjmp" guarantees that control is transfered directly to the handler *as soon as an exception is detected*.

On the other hand, some inconveniences do exist:

- designing the state object and its destructor/cleaner can be delicate. The "volatile" keyword must be used heavily to define its fields to ensure correct behaviour with "longjmp".
- the state object must be passed to *any transaction function*, which often constitutes a heavy requirement. And using global variables is simply a poor design decision unless you're 100% that your code is never going

to be used by multiple threads.

Indeed, for simple or short transactions, this scheme can be more painful than the paranoid one due to the "extra" code it requires. For example, here's our now famous `do_something` function implemented with it, we first need to write some support code:

```
    // define state object structure
    typedef struct
    {
      jmp_buf            jump_buffer;
      volatile char*     block1;
      volatile char*     block2;
      volatile char*     block3;

    } do_something_state;


   // define a simple custom allocator
    void*  checked_malloc(
              do_something_state*  state,
              size_t               size )
    {
      void*  p = NULL;

      if ( size > 0 )
      {
        p = malloc( size );
        if ( p == NULL )
        {
          // memory exhaustion,
          // jump to error handler
          longjmp( state->jump_buffer, 1 );
        }
      }

      return p;
    }
```

Our function code then becomes:

```
    // our function returns an error code
    int  do_something( ... )
    {
      volatile int        error;
      do_something_state  state;

      state.block1 = NULL;
      state.block2 = NULL;
      state.block2 = NULL;
      error        = ERR_OK;

      if ( setjmp( state.jump_buffer ) == 0 )
      {
```

```
      // allocate temporaries, notice that we
      // use a custom allocator to do that !!

      state.block1 = checked_malloc( &state, 1024 );
      state.block2 = checked_malloc( &state, 1024 );
      state.block3 = checked_malloc( &state, 1024 );

      .... // do something
    }
    else
    {
      // an exception was raised..
      error = ERR_MALLOC;
    }

    // cleanup
    if (state.block3) free( state.block3 );
    if (state.block2) free( state.block2 );
    if (state.block1) free( state.block1 );

    return error;

  }
```

That's certainly ***not*** tasty, but only because we wanted to catch exceptions within the function itself, and always return an error code. However, would the state had been designed and created before the function was called, we could have written it much more simply as:

```
  // our function does not return an error code
  void  do_something( do_something_state*  state, ... )
  {
    // allocate temporaries, notice that we use a custom
    // allocator to do that !!

     state->block1 = checked_malloc( &state, 1024 );
     state->block2 = checked_malloc( &state, 1024 );
     state->block3 = checked_malloc( &state, 1024 );

     .... // do something

     // cleanup, that's completely optional but it's
     // always a good idea to destroy temporaries as
     // soon as they're un-needed..
     free( state->block3 ); state->block3 = NULL;
     free( state->block2 ); state->block2 = NULL;
     free( state->block1 ); state->block1 = NULL;

     return;
  }
```

Which is considerably simpler. Most notably, this code *doesn't introduce jumps* to deal with exceptions, and *its structure is similar* to the unsafe code. Note also that the cleanup performed here is purely optional (since the function caller should always cleanup the state object).

We will thus conclude that the transaction scheme is only really useful when *only a few number of transactions exist*, and *when they need to perform relatively complex operations*. It otherwise doesn't scale very well to large projects.

Not surprisingly, it is used by the two popular LibPNG and LibJPEG libraries to implement their "load_image" and "save_image" operations. Other uses of this scheme is however pretty rare in other code with more complex high-level interfaces.

I.3. Structured Exception Handling (SEH):

Very few developers are aware that *structured exception handling (a.k.a. SEH) can be implemented in straight C*. **[1]**

There are even several portable implementations of SEH, all of them being based on `setjmp/longjmp`. They all provide special macros, like `TRY`, `CATCH`, `THROW`, etc ..., in order to mimic modern languages. These are generally used in "traditional" ways to implement "protected" statement blocks and exception handlers, as in the following example:

```
TRY
{
  ... the code that executes here is
  ... "protected", i.e. much like with
  ... transactions, any exception transfers
  ... control to what is between the CATCH and
  ... END_CATCH
}
CATCH(exc)
{
  // the following is only executed when an
  // exception was thrown. The exception code
  // is in "exc".
  switch (exc)
  {
    case ERR_MALLOC:
      fprintf( stderr, "out of memory\n" );
      break;

    default:
      // unknown exception, re-throw it
      RETHROW( exc );
  }
}
END_CATCH
```

Note that several ways to define the TRY, CATCH, etc.. macros exist , depending on the SEH sub-system **[2]**,. However, all implementations share the same basic principles:

- Each `TRY` call basically allocates a hidden, specialized object, named an "exception handler", which, among other things, contains a jump buffer for the "`setjmp`" instructions. The latter is called implicitely by the `TRY` macro.
- Exception handlers are stored in a singly linked list. Each TRY adds the new handler to the list. Each CATCH or END_CATCH is responsible to remove it from the list (if an exception didn't occur). This allows TRY..CATCH blocks to be *nested* (and is the reason for the "structured" term in SEH).
- Each `THROW` finds the current exception handler, unlink it, then uses `longjmp` with the jump buffer that it contains.

Since the macros are still clever ways to use `setjmp/longjmp`, one can rightly look at this as a simplified version of the transaction scheme, though with support for "nested" transactions. There is however a big difference here, since these macros do not deal with temporaries.

In other words, the exception handler itself is never used as a "transaction state". And *it's up to the developer to properly implement "history" and "rollback"* on top of them. This means that the SEH schemes, just like transactions, require some rather delicate design decisions from the programmer; and probably explains why they haven't been largely deployed in C programs.

Note also that when a program uses several threads with this scheme, each one of them must have its own list of exception handlers. This is normally implemented by using thread-local storage (TLS) objects to store the current exception handler, which are non-portable by default. This requires that the SEH implementation be aware of the current threading model used..

I.4 Cleanup Stack Exception Handling (CSEH):

Since manually implementing "state" management is usually very painful with transactions or SEH (e.g. it requires many ugly uses of the `volatile` keyword), the last technique presented in this paper was specifically designed to considerably ease this task. It is based on the idea that the "history" of any given operation can be implemented as a simple stack.

During program execution, a "cleanup stack" is maintained *for each running thread*. The stack is initially empty. When a new temporary is created or acquired, the program must "push" it on top of the cleanup stack, along with the address of a specific "cleanup function" that will be called to destroy or release it in case of exception.

For example, if a new memory block is allocated in the heap, it will be pushed with the address of the "free" function. If a mutex is locked, it can be pushed with the address of its "unlock" function, etc..

Note that a cleanup routine *shouldn't* raise an exception itself since it might be called within a THROW call. This would otherwise create complex double-fault cases that are simply too hard to deal with in a simple way.

Whenever the temporary becomes unuseful, or becomes part of the transient state of the program, it needs to be popped from the stack, and eventually cleaned up (i.e. destroyed/released).

This is normally achived by providing a small number of specific functions to developers:

- **void cleanup_push(item, cleanup_function, cleanup_data)**
  this function is used to push a new "item" on top of the cleanup stack. It stores on the stack's top-most slot the item pointer, as well as a "cleanup function" and its closure.

- **void cleanup_pop(item, keep_item)**
  this function pops the top-most item from the cleanup stack. If the `keep_item` boolean parameter is FALSE (the default), the item will be cleaned up (i.e. destroyed or released). The "item" parameter *must* correspond to the top-most stack element, for debugging reasons.

- **mark_t cleanup_mark(void)**
  which returns an opaque variable used to represent the stack's current height. The variable should only be called in a later `cleanup_unwind` call (see below).

- **void cleanup_unwind(mark)**
  which returns to a previous stack mark/height. this action automatically cleans up all items popped during the unwinding.

The definition of the SEH macros are also modified to perform the following additional operations:

- Each TRY calls `cleanup_mark` to record the current cleanup stack's height in the hidden exception handler.

- Each THROW or RETHROW first calls `cleanup_unwind` to clean all the temporaries that have been pushed since the TRY (it does so using the mark that was stored in the current exception handler). It then unlink the current handler, then calls `longjmp` to transfer control.

To make things clearer, we'll show how `do_something` works when written with this cleanup stack exception handling (CSEH) implementation. First we need some simple support routines like:

```
// allocate a new memory block, then push it
// on the cleanup stack. throws an exception
// in case of memory exhaustion
```

```
void*  malloc_push( size_t  size )
{
  void*  p = NULL;

  if ( size > 0 )
  {
    p = malloc(size);
    if ( p == NULL )
      THROW( ERR_ALLOC );
  }
  // use "free" as the cleanup routine,
  // the third parameter will be ignored
  cleanup_push( p, (cleanup_func_t)free, NULL );
}


// pop a block from the cleanup stack, then
// free it. we could define that as a macro,
// or simply invoke cleanup_pop directly in
// the code below..
void  pop_free( void*  block )
{
  // setting the second parameter to FALSE
  // ensures that the item is cleaned immediately
  cleanup_pop( block, FALSE );
}
```

Then, our function will look like:

```
void  do_something( .... )
{
  char  *block1, *block2, *block3;

  block1 = (char*) malloc_push( 1024 );
  block2 = (char*) malloc_push( 1024 );
  block3 = (char*) malloc_push( 1024 );

  ..... // do stuff
  .....

  pop_free( block3 );
  pop_free( block2 );
  pop_free( block1 );
}
```

The benefits of cleanup-stack based exception handling (a.k.a.CSEH) are important. Namely:

- the structure of safe code is *extremely* similar to the unsafe case. Compared to the original function implementation, we have only changed `malloc` to `malloc_push`, and `free` to `pop_free`. We have not introduced jumps to deal with errors, and the code is dead easy to read and understand.

- there is no need for a specific "state" object, since the cleanup stack is used to record the history of temporary creation and acquisitions. This means no additional and cumbersome structure type to define, no additional parameter to pass to functions, and no need to use the `volatile` keyword in most of the code.
- since the history is recorded in the cleanup stack, and not in the program's local variables, there is no reason to use the `volatile` keyword at all..

These features mean that writing reliable code with the CSEH scheme only needs *minor changes* to developer's design habits, and keeps the source simple to write, read and maintain !!

The next section contains a more detailed analysis of a CSEH implementations as well as descriptions of real-world implementations.

II. Analysis of CSEH Implementations

This section provides some important details regarding the performance costs and benefits of using a CSEH sub-system, as well as a list of real-world implementations.

II.1. Performance

Since temporaries need to be explicitely pushed and pop on the cleanup stack during program execution, using CSEH has a definite cost over "unsafe" code, and being able to quantify it is important.

This task is not easy however, because several factors affect the overall time of a program's execution: the number and frequency of push/pop pairs executed, as well as the *design* of these temporaries !.

Suppose for example that we need a function that reads a text file and returns a simple (singly linked) list of lines. We can do that with very simple code. Let's first define the relevant types:

```
// define string list type
typedef struct list_node_t  list_node_t;

struct list_node_t
{
  list_node_t*  next;
  char*         line;
};
```

```
typedef struct
{
  list_node_t*  first;
  int           count;

} list_t;
```

Now, let's write the string list finalizer, which will be used either for cleanup or for destroying/clearing the list

```
// define string list finalizer
void   list_done( list_t*  list )
{
  if ( list )
  {
    list_node_t  *node, *next;

    node = list->first;
    while (node)
    {
      next = node->next;
      free( node->line );
      free( node );
      node = next;
    }

    list->first = NULL;
    list->count = 0;
  }
}
```

We can now write our function very simply with:

```
// read file and builds new string list
void  list_read_file( list*   list,
                      file_t  file )
{
  list_node_t*   node;
  list_node_t**  pnode;
  char*          line;

  // initialise list
  list->count = 0;
  list->first = NULL;
  pnode       = &list->first;

  // push list on cleanup stack
  cleanup_push( list,
                (cleanup_func_t) list_done,
                NULL );

  // read file
  while ( !eof(file) )
  {
    line = file_read_new_line(file);
    if (line)
    {
      node = malloc_checked( sizeof(*node) );
      node->line = line;
      node->next = NULL;
      *pnode     = node;
      pnode      = &node->next;

      list->count++;
    }
  }

  // pop list before returning, keep it alive !!
  cleanup_pop( list, TRUE );
}
```

Notice that, independently of the number of lines that have been read from the file, this function only needs *one push/pop pair*. Each node is allocated through a simple and fast `malloc_checked` function. The cost of CSEH in this example is thus minimal, and this is also true for many other classes of containers (trees, hash tables, etc..).

OO programming tends to reduce the number of push/pop pairs considerably too. That's because the general rule about temporaries is that ***only "root" objects need to be moved to the stack***. Or, in other words, *an object that is "owned" by another one doesn't generally need to be pushed*.

We can thus *conjecture* that the cost of CSEH is minimal in real programs, but having some raw numbers is always a good idea. That's why a specific *micro-benchmark* has been written (called **c_bench.c**). It's a small C program whose purpose is to compare the timing of two functions that perform exactly the same things:

- **`do_test_1`**, a function that performs many allocations and releases of fixed-size blocks through unchecked calls to `malloc` and `free`.
- **`do_test_2`**, a variant, that uses `malloc_push` instead of `malloc`, `pop_free` instead of `free`, as well as a TRY..CATCH block to handle exceptions.

Note that the source of **c_bench.c** contains a simple but working implementation of a CSEH sub-system, complete with TRY, CATCH, THROW, etc.. macros. The code for `do_test_1` is simply:

```
#define  MAX_TEMPS   3000
#define  BLOCK_SIZE  32

  static void*  temps[ MAX_TEMPS ];

  static void  do_test_1( void )
  {
    int   n, i;
    for ( n = 1; n < MAX_TEMPS; n++ )
    {
      /* allocate 'n' temporary memory blocks */
      for ( i = 0; i < n; i++ )
        temps[i] = malloc( BLOCK_SIZE );

      /* now, release them (in reverse order) */
      for ( i = n-1; i >= 0; i-- )
        free( temps[i] );
    }
  }
```

The code for `do_test_2` is similar but simply renames the memory management functions to `malloc_push` and `pop_free`.

`c_bench.c` is highly portable, and the following table gives some timings for the execution of the two test functions with various compilers on the same machine:

comparing the performance of `malloc` and `malloc_push`

Timings are the average of 10 runs performed on a lighty loaded 233 MHz Mobile Pentium PC, 192 MB. All tests were run under Windows, except the last one

compiler	**do_test_1**	**do_test_2**	**difference**	**slowdown ratio**
Borland C++ 5.5	6.189s	7.921s	1.732s	28%
Visual C++ 6	8.932s	10.415s	1.483s	17%
Mingw GCC 2.95.2	20.299s	22.342s	2.043s	10%
Cygwin GCC 2.95.3	9.974s	11.376s	1.402s	14%
LCC-Win32	8.912s	10.245s	1.333s	15%
Linux GCC 2.95.4	5.89s	7.50s	1.61s	27%

These numbers correspond to more than 4 million alloc/free operations

Several important comments can be made from these numbers:

- *This microbenchmark doesn't reflect typical programs*. Instead, it really measures the performance of each compiler's C library memory manager. For example, the GCC allocator can be 2 to 3 times slower than other implementations on Windows, but beats all others on Linux. push/pop allocations under Borland C++ are faster than raw ones with Visual C++.

- *The `malloc_push` function is between 10% and 30% slower than raw `malloc`*, depending on the compiler and/or C library used.

- This means that if a program spends about 10% of its execution time doing memory management (a number that is probably higher than the reality of typical programs), then changing *all* allocations to use `malloc_push` and `pop_free` *would only incurr a penalty of 1% to 3% on the total running time*.

- However, *since only "root" temporaries need to be moved to the cleanup stack*, and that most memory allocations don't need a push/pop pair, we can claim, with high confidence, that ***the cost of the CSEH scheme on typical programs is simply negligible***

II.2. Requirements

a. Code size

Experience shows that implementing a CSEH runtime, i.e. the cleanup stack management functions, exceptions macros and routines, takes about 4 to 8 Kb of machine code. Indeed, most of our micro-benchmark's source code is devoted to implementing the CSEH sub-system, rather than the tests themselves.

The total amount of code taken by the runtime varies with processor type, as well as a few important details:

- A normal implementation would use dynamic allocation for the stack to support any number of temporaries, though still guarantee good performance for the push/pop routines. This is normally achieved by implementing the stack as a linked list of fixed-size "chunks" of slots, which further complexifies the code.

- In debug builds, the implementation must be paranoid in order to detect inconsistent stack states. These can be caused from developer errors (e.g. unordered push/pops) or complex bugs (i.e. dangling pointers).

- To support multiple-threads, some system specific routines must be provided to or implemented in the runtime system. They're used to retrieve thread-specific cleanup stacks and exception handler lists. This is normally done using thread-local storage (TLS) objects.

Note also that the implementation of certain runtime routines is delicate due to the use of dynamic allocation. For example, `cleanup_push` must deal with the case where there isn't any memory left to grow the cleanup stack.

Finally, note that this runtime's machine code size is fixed for all programs, and that code that uses it can be written in a simple and efficient way. Compare that to the paranoid scheme, where a _huge_ number of jumps are introduced in the source, which typically results in code size increase that is relative, instead of fixed.

b. Multiple threads

Designing code that can be run by multiple threads concurrently requires system-specific synchronisation primitives (like mutex, events, etc..) that are not ANSI-C portable. Likewise, each thread that might use the CSEH technique needs its own "exception context", where each context normally contains an exception handlers list as well as a cleanup stack.

On each TRY, CATCH, THROW, push or pop, the CSEH runtime must retrieve the current thread's exception context as quickly as possible. This is normally implemented through thread-local storage (TLS) objects. These are containers holding thread-specific values that are generally designed for very high speed (most implementations use non-blocking mutual exclusion, avoiding any kind of kernel call or other "expensive" processing).

However, this might affect the performance of the CSEH operations, including the push/pop functions. It's difficult to give any number to quantify this "penalty", since it is highly system-dependent (some platforms provide extremely fast TLS containers, others do not). However, we can still conjecture that the scarcity of push/pop operations during typical program operations should still keep the cost of CSEH minimal.

II.3. Real-World Implementations

A small number of real-world implementations of CSEH are already available and used with more or less extent.

KazLib

> **KazLib** is a package or reusable software modules, written by Kaz Kylheku, providing useful containers like lists, hash tables and dictionaries, as well as a portable CSEH implementation.
>
> It provides `try`, `catch` and `end_catch` macros, as well as `except_cleanup_push` and `except_cleanup_pop` functions to manage the cleanup stack.
>
> It's implementation is based on `setjmp/longjmp`. The library's source code includes routines to support thread-safety with Posix threads only.

The Symbian EPOC system

Symbian is the developer of EPOC, a C++ operating system specifically designed for embedded systems with drastic requirements, like cell phones or Psion PDAs.

EPOC provides its own thread sub-system and CSEH implementation, on which nearly all libraries and classes provided by the system to application developers rely. It is best described in **this specific document**.

Note that the EPOC implementation does not rely on native C++ exceptions for both portability and performance reasons. Since EPOC provides its own thread abstraction, they're supported by the CSEH sub-system.

The MLib

The **MLib** is sort of "general purpose" library designed specifically for embedded systems (it has some rather drastic requirements in this regard, it prohibits the use of static variables, for example). It is very similar in essence to something like the **GLib**, since it provides a range of useful containers (lists, hash tables, trees, etc..) and support routines. However, it was written with more selective requirements (it doesn't aborts the program in case of memory exhaustion) and provides a CSEH implementation on top of which the rest of the library is built.

By default, the MLib provides a fast single-threaded CSEH runtime. But client applications or libraries can provide their own synchronisation interface (including one to retrieve exception contexts) in order to automatically and transparently use it with multiple threads.

Conclusion

Let's summarize the various benefits and disadvantages of each one of the techniques surveyed in this paper. We'll compare them in terms of ease of use (from the developer's point of view), portability (in multi-thread contexts) as well as performance:

Paranoia

- easy to understand, and straightforward to implement, since it doesn't require runtime support
- highly portable, and supports multiple-threads by requiring only the most basic synchronisation primitives in most code.
- however, a _real_ pain when writing, reading and maintaining code. Most developers simply prefer to avoid it in their code.
- only used by code with strict reliability requirements. most C developers prefer to avoid it though (especially regarding memory exhaustion)

Transactions

- requires some understanding of the setjmp/longjmp functions, as well as the use of the "volatile" keyword for local variables of certain functions.
- highly portable too, since it relies on setjmp/longjmp which are provided by every ANSI C library.
- designing the transaction "state" object is delicate (and can be painful in terms of maintenance). However, writing functions that use it is very simple and pleasant.
- working, but rarely used (generally, when only a fews transactions exist and are completely identified).

structured exception handling

- requires some understanding of the TRY..CATCH..END_CATCH constructs, (not too difficult). Hides the use of "setjmp/longjmp" or any other code-jump implementation.
- highly portable for single-threaded programs, but requires some simple runtime support to manage the exception handlers list. Also requires system-specific TLS objects in order to support concurrent threads.
- very easy to use to manage (i.e. throw and catch) exceptions. However, the developer still needs to design "state" objects in order to implement proper "history/rollback".
- unfortunately, this technique is often used in "unsafe" ways, i.e. without implementing proper rollback in case of exceptions (especially in the C++ world).

cleanup stack exception handling
• requires some understanding of the TRY..CATCH..END_CATCH constructs, (not too difficult), as well as the push/pop operations (not too difficult too). • highly portable for single-threaded programs, but requires some complex runtime support to manage the exception contexts (handlers list and cleanup stack). Also requires system-specific TLS objects in order to support concurrent threads. • very easy to use, both to manage exceptions and "history". actually, "safe" code using the CSEH technique is highly similar to "unsafe" one. It is thus significantly easier to write, read and maintain.. • not widely used until now, but already tested on a wide scale (see Symbian/EPOC).

It should be obvious from this table that even if the CSEH scheme hasn't been very used until now, it is *vastly superior* to other design techniques in terms of ease of use and that most developers could become aware of it and start using it. Of course, that will not be possible until some good runtime support for CSEH is available in one of the popular toolkits used by C or C++ developers today.

The document's author hopes that this paper will stir interest in this technique and motivate enough people to introduce such features in the libraries and programs they write..

David Turner (**david@freetype.org**)

Annex A : the `setjmp` and `longjmp` instructions

`setjmp` and `longjmp` are two portable ANSI C instructions that are used to transfer control *out of C function* (unlike the `goto` statement). You need to

```
#include <setjmp.h>
```

to be able to use it in your programs.

Base mechanism:

First of all, `setjmp`" records the current *processor state* (i.e. instruction pointer and register state) in a system-specific structure known as a "jump buffer", of type `jmp_buf`. The function normally returns immediately, with the value 0.

Now, "`longjmp`" is used with a `jmp_buf` structure that must have been previously initialised with `setjmp`. It simply *directly transfers control* to the IP stored in the jump buffer. This is done *without stack unwinding or checking*. once transfer is controlled, the corresponding `setjmp` function returns with a non-0 value (normally provided by the `longjmp`).

As an example, here is some code:

```
static jmp_buf   jump_buffer;

void  f1( int  x )
{
  // print something
  printf( "the value of x is %d\n", x );

  // transfer control, the 1 is unimportant
  longjmp( jump_buffer, 1 );

  // never executed
  printf( "the value of 2*x is %d\n", 2*x );
}


void  f2( int  x )
{
  // never returns
  f1( x );

  // never executed
  printf( "the value of x^2 is %d\n", x*x );
}


void  main( void )
{
  int  x = 509;

  if ( setjmp( jum_buffer ) == 0 )
  {
    // print header
    printf( "beginning !!\n" );
```

```
      // never returns
      f1( x );

      // never executed
      printf( "what ???\n" );
    }
    else
    {
      // executed after the "longjmp" in f1
      printf( "branch taken !!\n" );
    }

    // print footer
    printf( "exiting !!\n" );
  }
```

Here, the `main` function calls `f2`, which itself calls `f1`

Special considerations (the `volatile` keyword)

Care must be taken with local variables in functions using `setjmp()`. Because most optimizing compilers store locals in registers, and since the `setjmp/longjmp` functions store and load register values in the jump buffer, some hideous bugs can happen. For example, consider the following code:

```
  {
    int      counter = 0;
    jmp_buf  jump_buffer;

    printf( "%d ", counter );
    counter++;

    if ( setjmp( jump_buffer ) == 0 )
    {
      printf( "%d ", counter );
      counter++;
      longjmp( jum_buffer, 1 );
    }

    printf( "%d\n", counter );
  }
```

You would expect this function to print `"0 1 2"`, but with most optimizing compilers, you'll get `"0 1 1"` instead. That's because the `counter` variable was optimised in a register instead of a stack slot. When `setjmp` is called, it saves all register values within the "`jump_buffer`", including the current `counter` value, which is 1. When `longjmp` is called, it restores all register values from the buffer.

The second counter increment is thus lost..

A simple way to deal with this is to use the `volatile` storage specifier to define your local variables. This indicates to the compiler that you don't want this variable to be "optimized" in a register, and avoids the bug..

Just replace the second line above with:

```
volatile int  counter = 0;
```

And everything will work well. Note that since `TRY`, `CATCH` and `THROW` macros in SEH and CSEH schemes are simple ways to encapsulate calls to `setjmp/longjmp`, you'll need to take care of this problem in every function that declares local variables and uses `TRY..CATCH` blocks..

Footnotes

[1] That's probably because most modern programming languages that support them use a sophisticated implementation that requires some special magic to happen at *compile time*: Basically, A C++ or Ada compiler must emit, for each function that may fail, some automatically-generated cleanup code and exception handler tables within the program's machine code. Of course, this technique simply *cannot* be coded in C.

[2] Note that Microsoft has introduced specific extensions to the C language in Visual C++ in order to be able to manage native exceptions on Windows. This is done by using new keywords (like "`__try`", "`__catch`" or "`__finally`"), which are *not* macros, and do not rely on `setjmp/longjmp`.

These are compiler-specific extensions and shouldn't be used in any portable programs (even if other Windows compiler now provide the same extensions).