1. Introduction

XXL is a library for C and C++ that provides exception handling and asset management. Asset management is integrated with the exception handling mechanism such that assets may be automatically cleaned up if an exception is thrown, which allows for much simplified program structure with respect to error handling.

By allowing XXL to track assets and using its exception handling features, the programmer no longer has to check error conditions on every function call and cleanup the assets on failure because XXL does the work. For example, consider the following code that allocates three 4K buffers:

```
static pthread_mutex_t buffer_lock = PTHREAD_MUTEX_INITIALIZER;

int allocate_buffers(char **first, char **second, char **third)
{
    pthread_mutex_lock(&buffer_lock);
    if (!(*first = (char *)malloc(4096)))
    {
        pthread_mutex_unlock(&buffer_lock);
        return 0;
    }
    if (!(*second = (char *)malloc(4096)))
    {
        free(*first);
        pthread_mutex_unlock(&buffer_lock);
        return 0;
    }
    if (!(*third = (char *)malloc(4096)))
    {
        free(*second);
        free(*first);
        pthread_mutex_unlock(&buffer_lock);
        return 0;
    }
    pthread_mutex_unlock(&buffer_lock);
    return 1;
}
```

Using XXL's asset management and exception handling services, the code can be simplified a great deal:

```
static pthread_mutex_t buffer_lock = PTHREAD_MUTEX_INITIALIZER;

static void alloc_or_throw(size_t nbytes)
{
    void *ptr;

    if (!(ptr = malloc(nbytes)))
        XXL_THROW_ERROR(ENOMEM, NULL);
    XXL_ASSET_SAVE(ptr, free, NULL, XXL_ASSET_PROMOTE);
    return ptr;
}

int allocate_buffers(char **first, char **second, char **third)
{
    int result = 1;
```

```
    XXL_TRY_BEGIN
    {
        pthread_mutex_lock(&buffer_lock);
        XXL_ASSET_SAVE(&buffer_lock, pthread_mutex_unlock, NULL,
                       XXL_ASSET_TEMPORARY);
        *first = (char *)alloc_or_throw(4096);
        *second = (char *)alloc_or_throw(4096);
        *third = (char *)alloc_or_throw(4096);
    }
    XXL_EXCEPT
    {
        result = 0;
    }
    XXL_TRY_END;

    return result;
}
```

The benefits to using XXL's facilities are quickly realized by any C programmer that has had to perform a series of operations requiring additional cleanup in the event of an error each step of the way. XXL allows the programmer to spend less time worrying about error conditions, and to spend more time working on the "real" code instead. Error handling is notoriously prone to further errors—forgetting to clean up some asset in some cases, but not in others can often lead to program defects that are difficult to track down.

2. Handling Exceptions

The semantics of XXL's exception handling are not unlike those found in languages that have built-in exception handling, such as C++, Java, or Python. The structure of an exception handler is the same with XXL as it is in languages that provide native exception handlers. All exception handlers begin with a block of code that may throw an exception. For example, in C++:

```
try
{
    // Do something here that may cause an exception to be thrown
}
```

The initial block of code that may throw an exception is always executed in its entirety unless an exception is thrown, in which case the code flow is redirected to an appropriate exception handler. Most languages allow arbitrary types to be thrown as exceptions, and the exception handlers catch the exceptions based on type rather than value. Unfortunately, due to limitations imposed by C, XXL can only throw integer values as exceptions, and exceptions are caught based on their value.

Some languages such as Java and Python support the ability to include a block of code that will always be executed, regardless of whether an exception was thrown or not. This feature is not included in C++'s exception handling feature set[1], but XXL does support it. Such a block of code is usually referred to as a "finally block," because the keyword used to delimit it is usually "finally."

As with most languages that support exception handling natively, the ordering of code blocks in an exception handler is significant in XXL's implementation. The "try" block must occur first, followed by "catch" blocks, an optional "except" block, and the optional "finally" block must always be last. The general structure is thus:

```
XXL_TRY_BEGIN
{
}
XXL_CATCH(...)
```

[1] There is no provision in the C++ standard for the feature; however, some C++ compilers such as Microsoft Visual C++ do support it as an extension. The most popular compiler for Unix systems, GCC, does not support the feature, but patches are available that can be applied to the compiler to add support.

```
{
}
XXL_EXCEPT
{
}
XXL_FINALLY
{
}
XXL_TRY_END;
```

Any number of XXL_CATCH blocks may be present as required, but only a single XXL_EXCEPT block may be present that must follow all XXL_CATCH blocks. While the current implementation can handle multiple except XXL_FINALLY blocks and have them behave as one might expect, future versions may not, so you should never include any more than a single XXL_FINALLY block.

Finally, while other exception handling implementations may handle prematurely leaving an exception handler block gracefully, XXL will not. You must never use `return`, `goto`, C++ exceptions, `break`, `continue`, or `setjmp()`/`longjmp()` to leave an XXL exception handler code block. If you must leave an exception handler code block, use `XXL_LEAVE()`, which will transfer control to the code immediately following the exception handler's XXL_TRY_END.

XXL_TRY_BEGIN

This macro begins an exception handler. Any code that immediately follows it will always be executed in its entirety unless an exception is thrown. If no exception is thrown, control will be transferred to the XXL_FINALLY block if one is present; otherwise, control will be transferred to the code immediately following XXL_TRY_END, which must be present within the same scope as XXL_TRY_BEGIN to terminate the exception handler.

If XXL_ENFORCE_PREFIX is not defined, TRY is a synonym for XXL_TRY_BEGIN.

XXL_CATCH(code)

This macro begins an exception handler "catch" block that will only be executed if the specified exception is thrown. The exception to catch is specified by substituting `code` for the integer value of the exception. For example, if ENOMEM is thrown when an out of memory condition occurs, `XXL_CATCH(ENOMEM)` would be the proper way to catch the exception.

An exception handler may have no `XXL_CATCH()` blocks, or it may have as many as necessary to catch exceptions that may be thrown. You should not have multiple `XXL_CATCH()` blocks for the same exception code. If you do, only the first one will be executed. The ordering of `XXL_CATCH()` blocks is not important.

If XXL_ENFORCE_PREFIX is not defined, CATCH is a synonym for XXL_CATCH.

XXL_EXCEPT

This macro begins a special exception handler "catch" block that will catch any exception thrown that is not otherwise handled by `XXL_CATCH()`. If this macro is used, it must occur after all `XXL_CATCH()` blocks, but before an XXL_FINALLY block if one is also present. An exception handler may have only a single XXL_EXCEPT block.

If XXL_ENFORCE_PREFIX is not defined, EXCEPT is a synonym for XXL_EXCEPT.

XXL_FINALLY

This macro begins a block of code within an exception handler that will be always be executed, regardless of whether an exception is thrown. If no exception is thrown, it will be executed immediately after the block of code immediately following XXL_TRY_BEGIN. If an exception is thrown, it will be executed after the exception's handler is executed.

XXL_FINALLY blocks are optional, but if one is present, only one should be used. In other words, do not include multiple XXL_FINALLY blocks in a single exception handler.

If XXL_ENFORCE_PREFIX is not defined, FINALLY is a synonym for XXL_FINALLY.

XXL_TRY_END

This macro ends an exception handler. For every XXL_TRY_BEGIN, there must be a corresponding XXL_TRY_END within the same scope. While most languages with native exceptions do not require such an ending to exception handlers, limitations imposed by the implementation in C require it. Unlike the other exception handler block delimiting macros, you must always follow XXL_TRY_END with a semi-colon.

If XXL_ENFORCE_PREFIX is not defined, END_TRY is a synonym for XXL_TRY_END.

3. Throwing Exceptions

As noted in the previous section, XXL exceptions are always integer values rather than arbitrary data types. As such, exception handlers catch exceptions by their value. We recommend that you reserve the first 256 values (0 through 255) for standard C error codes such as ENOMEM, EINVAL, and so on.

XXL_THROW_ERROR(code, data)

This macro will cause the exception specified by code to be thrown. The code must be a signed integer value of the type int. When an exception is thrown, an arbitrary data pointer may be attached to the exception, which is of type void *. The current exception code may be retrieved from within an XXL_CATCH() or XXL_EXCEPT code block using XXL_EXCEPTION_CODE(). The data pointer associated with the current exception may be retrieved using XXL_EXCEPTION_DATA().

When an exception is thrown, the first exception handler "catch" block that is appropriate will catch the exception. If the current exception handler has no suitable "catch" block, the exception will be propagated up to the previous exception handler. This process is repeated until a suitable "catch" block is found. If no "catch" block can be found for an exception, XXL will terminate the program with a suitable error message.

If you pass a data pointer along with an exception, take care that you do not pass a pointer to data on the stack within the current function if the exception may be propagated up to another handler outside of the current scope; otherwise, the receiving "catch" block will be given a dangling pointer if it uses XXL_EXCEPTION_DATA() to retrieve the exception's data pointer.

If XXL_ENFORCE_PREFIX is not defined, THROW is a synonym for XXL_THROW_ERROR.

XXL_RETHROW_ERROR()

From within a "catch" block, this macro will re-throw the current exception. The current exception handler will not be considered for handling the exception. It will be automatically propagated up to the previous exception handler, and the same rules for finding a suitable "catch" block to handle the exception will be followed as with XXL_THROW_ERROR().

If XXL_ENFORCE_PREFIX is not defined, RETHROW is a synonym for XXL_RETHROW_ERROR.

XXL_LEAVE()

This macro will prematurely leave an exception handler code block, transferring control to the code immediately following the current handler's XXL_END_TRY. If you must leave an exception handler code block prematurely, this is the only way you should do so. Do not ever use return, goto, break, continue, C++ exceptions, or setjmp()/longjmp() to leave an exception handler code block.

XXL_RETRY()

This macro will cause control to be transferred back to the beginning of the current exception handler's `XXL_TRY_BEGIN` code block. If it is used from within a "catch" block, it will not return control to the point where the exception was thrown.

XXL_EXCEPTION_CODE()

This macro will retrieve the current exception's value. It should only be used from within a "catch" block. If it is used anywhere else, its result is undefined.

If `XXL_ENFORCE_PREFIX` is not defined, `EXCEPTION_CODE` is a synonym for `XXL_EXCEPTION_CODE`.

XXL_EXCEPTION_DATA()

This macro will retrieve the data pointer associated with the current exception. It should only be used from within a "catch" block. If it is used anywhere else, its result is undefined.

If `XXL_ENFORCE_PREFIX` is not defined, `EXCEPTION_DATA` is a synonym for `XXL_EXCEPTION_DATA`.

XXL_EXCEPTION_FILE()

This macro will retrieve the source file from whence the current exception was thrown. It should only be used from within a "catch" block. If it is used anywhere else, its result is undefined.

If `XXL_ENFORCE_PREFIX` is not defined, `EXCEPTION_FILE` is a synonym for `XXL_EXCEPTION_FILE`.

XXL_EXCEPTION_LINE()

This macro will retrieve the source line number from whence the current exception was thrown. It should only be used from within a "catch" block. If it is used anywhere else, its result is undefined.

If `XXL_ENFORCE_PREFIX` is not defined, `EXCEPTION_LINE` is a synonym for `XXL_EXCEPTION_LINE`.

4. Asset Management

In addition to providing support for exceptions in C, XXL provides asset management facilities that are tightly integrated with its exception handling mechanism. Assets are arbitrary data pointers that have a callback function associated with them. The callback function is called when the asset must be cleaned up.

Assets can be permanent, promotable, temporary, or automatic. The asset manager will never clean up permanent assets. In fact, the asset manager will never even record them. Their purpose is primarily as a convenience for the programmer. That way, the programmer may write a function such as the following:

```
void *safe_malloc(size_t nbytes, xxl_assettype_t type)
{
    void *ptr;

    if (!(ptr = malloc(nbytes)))
        XXL_THROW_ERROR(ENOMEM, (void *)nbytes);
    XXL_ASSET_SAVE(ptr, free, NULL, type);
    return ptr;
}
```

By providing a permanent asset type, the programmer is freed from having to check the type before calling `XXL_ASSET_SAVE()`, instead allowing `XXL_ASSET_SAVE()` to perform the check.

Promotable assets are cleaned up only when an exception is thrown. If no exception is thrown, the record of the asset is erased and the asset effectively becomes permanent. Demotable assets are the exact opposite

of promotable assets; they are cleaned up only if no exception is thrown. Temporary assets are cleaned up regardless of whether an exception is thrown. Finally, automatic assets are presently treated the same as permanent assets except that the asset manager does record them. Automatic assets are intended for use by C++ classes that are allocated on the stack, but additional support for automatic assets is not present in this release.

XXL_ASSET_BLOCK_BEGIN

This macro begins a new asset management block. Note that `XXL_TRY_BEGIN` will also begin a new asset management block. Assets that are saved within the bounds of an asset management block will be cleaned up as appropriate when the block is ended. Asset management blocks may be—and frequently are—nested.

XXL_ASSET_BLOCK_END

This macro ends an asset management block. Note that `XXL_TRY_END` will also end the asset management block begun by its corresponding `XXL_TRY_BEGIN`. When an asset management block is ended normally, only temporary assets will be cleaned up. Promotable assets will be promoted to permanent assets. If an exception is thrown from within an asset management block, promotable assets will also be cleaned up.

XXL_ASSET_SAVE(asset, callback, arg, type)

This macro saves an asset within the current asset management block. The asset is an arbitrary data pointer that is passed to its associated clean up callback function. In addition, a second arbitrary pointer may be specified that will be passed unmolested to the callback function. Finally, every asset has a type: permanent, promotable, temporary, or automatic.

`asset`
: This is the asset to be saved.

`callback`
: This is the callback function that should be called to clean up the asset if necessary. The callback function should accept two arguments, both pointers to `void`. The first argument passed will be the asset that is to be cleaned up, and the second argument will be the data pointer specified as `arg` to `XXL_ASSET_SAVE()`.

`arg`
: This is an arbitrary data pointer that will be passed unmolested to the callback function when the asset needs to be cleaned up.

`type`
: This is the type of asset. Valid values are `XXL_ASSET_PERMANENT`, `XXL_ASSET_PROMOTE`, `XXL_ASSET_TEMPORARY`, or `XXL_ASSET_AUTO`.

XXL_ASSET_UPDATE(old, new)

This macro updates an asset's data pointer. It is most useful for memory pointers that are reallocated using `realloc()` or a similar function such as `GlobalReAlloc()` or `LocalReAlloc()` on Windows. It is important to note that assets can only be updated within the current thread. In other words, if the asset is saved in another thread as well (generally not a good idea), it will not be updated by `XXL_ASSET_UPDATE()` on a different thread. An example of this function's usage would be:

```
void *safe_realloc(void *old_ptr, size_t nbytes)
{
    void *new_ptr;

    if (!(new_ptr = realloc(old_ptr, nbytes)))
        XXL_THROW_ERROR(ENOMEM, (void *)nbytes);
    XXL_ASSET_UPDATE(old_ptr, new_ptr);
```

```
        return new_ptr;
    }
```

XXL_ASSET_RELEASE(asset, mode)

This macro releases an asset's record depending on the specified mode, which can be one of `XXL_ASSET_ALL`, `XXL_ASSET_CURRENT`, or `XXL_ASSET_FIRST`. Regardless of the mode, only assets stored in the current thread's asset management stack will be consulted for removal. If the asset is saved in another thread as well (generally not a good idea), it will not be removed by `XXL_ASSET_RELEASE()` on a different thread regardless of the mode.

`XXL_ASSET_ALL`
: Asset records matching the specified asset will be removed from all asset management blocks in the current thread. No cleanup functions will be called.

`XXL_ASSET_CURRENT`
: Asset records matching the specified asset will be removed only from the current asset management block. No cleanup functions will be called.

`XXL_ASSET_FIRST`
: Only the first asset record matching the specified asset will be removed, regardless of whether it is in the current asset management block or not. No cleanup functions will be called. The first asset record will be the most recently saved record.

5. Convenience Functions

Version 0.9.5 adds a sizable set of convenience functions. These functions are wrappers around common functions such as memory allocation, file opening, and mutexes. Also included are asset clean up functions that may be used with or without the other convenience functions. Since all of the provided convenience functions are simply wrappers around existing, well known functions, full documentation is not provided for each here. Instead, this section simply contains a list of the functions, their signatures, and the functions they wrap.

Memory Allocation Functions

On all platforms, wrapper functions are provided for the standard `malloc`, `realloc`, and `free` functions that are part of the standard C runtime library. On Windows, additional wrapper functions are provided for the local and global heap memory allocation functions.

In addition, three macros are provided that can be used to select the memory allocation functions that are appropriate for the platform XXL is being used on. On Unix, the standard C wrappers are used. On Windows, the process heap is used unless `XXL_USE_GLOBALHEAP` or `XXL_USE_LOCALHEAP` are defined, in which case the global or local heaps are used as appropriate. If both are defined, the global heap takes precedence over the local heap. The memory allocation wrapper macros are `XXL_MALLOC`, `XXL_REALLOC`, and `XXL_FREE`, all of which require arguments corresponding to `xxl_malloc`, `xxl_realloc`, and `xxl_free` signature, respectively.

XXL Convenience Function	Wrapped Function	Clean Up Function
`void *xxl_malloc(size_t, xxl_assettype_t);`	`malloc`	`xxl_cleanup_ptr`
`void *xxl_realloc(void *, size_t);`	`realloc`	
`void  xxl_free(void *);`	`free`	
`HGLOBAL xxl_GlobalAlloc(UINT, DWORD, xxl_assettype_t);`	`GlobalAlloc`	`xxl_cleanup_HGLOBAL`
`HGLOBAL xxl_GlobalReAlloc(HGLOBAL, DWORD, UINT);`	`GlobalReAlloc`	
`HGLOBAL xxl_GlobalFree(HGLOBAL);`	`GlobalFree`	
`void *xxl_HeapAlloc(HANDLE, DWORD, DWORD, xxl_assettype_t);`	`HeapAlloc`	`Xxl_cleanup_HeapPtr`
`void *xxl_HeapReAlloc(HANDLE, DWORD, void *, DWORD);`	`HeapReAlloc`	

`void *xxl_HeapFree(HANDLE, void *);`	`HeapFree`	
`HLOCAL xxl_LocalAlloc(UINT, UINT, xxl_assettype_t);`	`LocalAlloc`	`xxl_cleanup_HLOCAL`
`HLOCAL xxl_LocalReAlloc(HLOCAL, UINT, UINT);`	`LocalReAlloc`	
`HLOCAL xxl_LocalFree(HLOCAL);`	`LocalFree`	

File Access Functions

Wrapper functions are provided for the standard C runtime file access functions. In particular, `fopen` and `fstream`. The Windows `CreateFile` function is not wrapped, primarily due to its already overloaded signature. A generic cleanup function for Windows handles is provided, however.

XXL Convenience Function	Wrapped Function	Clean Up Function
`FILE *xxl_fopen(const char *, const char *, xxl_assettype_t);`	`fopen`	`xxl_cleanup_FILE`
`int xxl_fclose(FILE *);`	`fclose`	
`int xxl_open(const char *, int, mode_t, xxl_assettype_t);`	`Open`	`xxl_cleanup_fd`
`int xxl_close(int);`	`close`	`xxl_cleanup_HANDLE`

Socket Access Functions

Wrapper functions are provided for socket access functions. The wrapped functions map to standard socket functions. On Windows, linking against `ws2_32.lib` is required.

The Unix variations:

XXL Convenience Function	Wrapped Function	Clean Up Function
`int xxl_socket(int, int, int, xxl_assettype_t)`	`socket`	`xxl_cleanup_socket`
`int xxl_shutdown(int, int);`	`shutdown`	
`int xxl_closesocket`	`close`	

The Windows variations:

XXL Convenience Function	Wrapped Function	Clean Up Function
`SOCKET xxl_socket(int, int, int, xxl_assettype_t);`	`socket`	`xxl_cleanup_socket`
`int xxl_shutdown(SOCKET, int);`	`shutdown`	
`int xxl_closesocket(SOCKET);`	`closesocket`	

Mutex Functions

Wrapper functions are provided for manipulating mutex objects. For example, when a mutex is acquired/locked, a clean up function is saved with the mutex to unlock it as specified by the asset type (temporary, promotable, and so on). On Unix systems, wrappers are provided for pthreads mutex objects. On Windows systems, wrappers are provided for the Win32 mutex objects.

The Unix variations:

XXL Convenience Function	Wrapped Function	Clean Up Function
`int xxl_lock(pthread_mutex_t *, xxl_assettype_t);`	`pthread_mutex_lock`	`xxl_cleanup_lock`
`int xxl_unlock(pthread_mutex_t *);`	`pthread_mutex_unlock`	

The Windows variations:

XXL Convenience Function	Wrapped Function	Clean Up Function
`BOOL xxl_lock(HANDLE, xxl_assettype_t);`	`WaitForSingleObject`	`xxl_cleanup_lock`
`BOOL xxl_unlock(HANDLE);`	`ReleaseMutex`	