```
  if (GoodName<100>::validate("Corey Duffle")) {
    cout << "'Corey Duffle' is a valid name.\n";
  }

  if (!GoodName<100>::validate("Sir Chumley the 5th")) {
    cout << "'Sir Chumley the 5th' is NOT a valid name.\n";
  }
}
```

4.5.10 Example Resolved

The university has identified three sources of input to their (very simple) ERP system:

1. a database system
2. GET parameters from HTML forms
3. POST parameters from HTML forms

The university has written a utility library to read GET/POST parameters that allows the developers to easily specify a validity checking routine to use when reading GET/POST parameters. The developers are using a simple static analysis tool to ensure that all reads of GET/POST parameters occur only through the utility library. They have instituted formal code reviews of the input validation checking routines to ensure that all input validation criteria are implemented correctly.

The university is using a third-party database abstraction library that sanitizes all provided strings that are to be used in the creation of SQL queries and provides some basic sanity checking of the results of SQL queries.

4.5.11 Known Uses

Many web frameworks and languages and general programming libraries provide support for performing input validation and sanitization. Frameworks with known input validation support include

- Ruby on Rails
- Java Struts
- Pylons
- Django

The Secure XML-RPC Server Library uses the Input Validation secure design pattern.

4.6 Resource Acquisition Is Initialization (RAII)

4.6.1 Intent

The intent of the RAII pattern is to ensure that system resources are properly allocated and deallocated under all possible program execution paths. RAII ensures that program resources are properly handled by performing resource allocation and deallocation in an object’s constructor and destructor, removing the need for external users of an object to handle the allocation and deallocation of the object’s resources.

4.6.2 Motivation

Typically every resource that is used must be released in a timely manner. This is necessary to prevent resource exhaustion. It is also important not to release resources while they are still being used. This often has fatal consequences. For instance, usage of memory that has been previously freed is widely considered a security flaw because many memory allocation systems re-use freed memory when further memory is requested. The usage of freed memory might consequently overwrite data that was stored in memory requested after the original memory was freed.

Furthermore, the maintenance of when to free resources often becomes a daunting task due to large numbers of reserved resources. Unless a resource's lifetime is planned in the software design, it is difficult to determine when the resource is no longer necessary and may be released.

4.6.3 Example

One example of the use of the RAII pattern is a program that allocates memory at the beginning of a function and frees the memory before the function exits. This includes freeing the memory under alternate control flows. For instance, if the function throws an exception or halts the program, it still frees the memory first. Another example of the use of the RAII pattern is an object that opens a network connection when it is constructed and closes the network connection when it is destroyed.

Similarly, an object might open a file when it is constructed. In this case the object must close the file in its destructor. If the opening of the file is optional, the destructor assumes the responsibility for closing the file if and only if it has been opened.

4.6.4 Applicability

RAII applies to any system that uses a resource that must be acquired and subsequently released. Such resources include regions of memory, opened file descriptors, and network resources, such as open sockets.

The pattern is useful when the amount of available resources is finite and limited and when failing to release acquired resources yields resource exhaustion and denial of service.

4.6.5 Structure

The structure of the RAII secure design pattern is relatively straightforward. In the common code executed at the start of the lifetime of an object (commonly in the object's constructor in object-oriented languages), allocate resources. In the common code executed at the end of the lifetime of an object (commonly the object's destructor in object-oriented languages), deallocate resources.

4.6.6 Participants

The participants in the RAII pattern are

- the object making use of system resources
- the system resources

4.6.7 Consequences

RAII enforces automatic resource management, in that a resource is acquired only by the object or function that needs it, and the resource is never left unfreed after the object's lifetime. The program might run more slowly with RAII than it might run with an alternate resource management scheme, such as garbage collection. Such comparisons are highly implementation-dependent.

4.6.8 Implementation

RAII enforces automatic resource management, in that a resource is acquired only by the object or function that needs it. When an object manages a resource, the object typically allocates the resource in its constructor and releases the resource in its destructor. The object's destructor must also be invoked when the object itself is no longer required. But this is itself another instance of RAII, where the object that manages a resource is itself another resource and must be managed by another object or function.

When a function manages a resource, the function typically allocates the resource during its execution (often near the beginning) and releases the resource before it returns. The developer must be aware of all forms of abnormal exit of the function, such as exceptions, and must ensure the resource is released upon any exit venue. That is, if the function calls a subfunction that throws an exception, the function must catch the exception and release the resource before handling the exception or rethrowing it.

For more implementation details, see the following CERT secure coding guidelines:

- For C++, see FIO42-CPP, "Ensure files are properly closed when they are no longer needed for file-based RAII" [CERT 2009b]
- For C, see MEM00-C, "Allocate and free memory in the same module, at the same level of abstraction for memory-based RAII" [Seacord 2008]
- For Java, see FIO34-J, "Ensure all resources are properly closed when they are no longer needed for network-based RAII" [CERT 2009c]

4.6.9 Sample Code

RAII is most prevalent in C++ because an automatic variable object in C++ will have its destructor called when its scope terminates, either normally or through a thrown exception.

The following RAII class is a lightweight wrapper of the C standard library file system calls.

```
#include <cstdio>
#include <stdexcept> // std::runtime_error
class file {
public:
  file (const char* filename) : file_(std::fopen(filename, "w+")) {
    if (!file_)
      throw std::runtime_error("file open failure");
  }

  ~file() {
    if (0 != std::fclose(file_)) { // need to flush latest changes?
      // handle it
    }
  }

  void write (const char* str) {
    if (EOF == std::fputs(str, file_))
      throw std::runtime_error("file write failure");
  }

private:
  std::FILE* file_;

  // prevent copying and assignment; not implemented
  file (const file &);
  file & operator= (const file &);
};
```

Class `file` can then be used as follows:

```
void example_usage() {
  file logfile("logfile.txt"); // open file (acquire resource)
  logfile.write("hello logfile!");
  // continue using logfile ...
  // throw exceptions or return without worrying about closing the
  // log; it is closed automatically when logfile goes out of scope
}
```

This works because the class `file` encapsulates the management of the `FILE*` file handle. When objects `file` are local to a function, C++ guarantees that they are destroyed at the end of the enclosing scope (the function in the example), and the `file` destructor releases the file by calling `std::fclose(file_)`. Furthermore, `file` instances guarantee that a file is available by throwing an exception if the file could not be opened when creating the object.

4.6.10 Known Uses

The BOOST library provides the **`boost::shared_ptr`**, which is also marked for inclusion in the new C++0x standard. It is a smart pointer that uses reference counting to manage pointed-to objects, and it guarantees that the referenced objects are destroyed when the **`shared_ptr`** is destroyed.

The Secure XML-RPC Server Library uses the RAII secure design pattern to implement the Clear Sensitive Information secure design pattern.