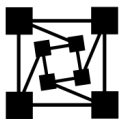
ParaSoft®

insure++®

Insure++: An Automatic Runtime Error Detection Tool



ParaSoft®

1. Introduction

Insure++ is an automatic runtime error detection tool for C/C++ applications that uncovers problems such as memory corruption, memory leaks, pointer errors, and I/O errors. By using a unique set of patented technologies, Insure++ thoroughly examines and tests the code from inside and out, then reports errors and pinpoints their exact location. Insure++ also performs coverage analysis, clearly indicating which sections of the code were tested, and facilitates memory optimization, displaying how the program uses memory in real-time. By integrating Insure++ into your development environment, developers can save weeks of debugging time and prevent costly crashes from affecting your customers.

Unfortunately, development tools like Insure++ usually aren't called into action until the end of a software project, when particularly difficult bugs causing erratic behavior cannot be found. The typical crisis cycle goes something like this: developers exhaust all options searching for the source of a bug, they give up, they use Insure++, Insure++ finds the bug, the developers fix the bug, and then move on until the next crisis hits.

This crisis cycle could be broken, resulting in less aggravation and less time spent debugging, if Insure++ was incorporated into the software development process earlier. Insure++ has been used on C/C++ applications with hundreds of thousands of lines of code; multi-process applications, programs distributed over hundreds of workstations, operating systems, and compilers have been validated with Insure++. This paper describes Insure++'s modes of operation, how to implement Insure++ in the development process, and the types of errors Insure++ uncovers.

2. How Insure++ Works

Insure has two modes of operation that provide developers with varying degrees of error checking: Chaperon for a quick check and Source Code Instrumentation for deep error checking. This gives developers a large amount of flexibility when it comes to maximizing their development and debugging time. In both modes, when Insure++ finds a problem, it provides a complete diagnosis, including the names of related variables, the line of source code containing the error, a description of the error, and a stack trace.

2.1 Chaperon

Insure++'s latest technology--Chaperon--works with existing executable programs. Chaperon does not require any recompiling and relinking, or changing of environment variables. Though less intensive than Source Code Instrumentation mode, Chaperon is more expedient. It checks all data memory references made by a process, whether in compiled code, language support routines, or shared or archived libraries. It also detects

and reports reads of uninitialized memory, reads or writes that are not within the bounds of allocated blocks, and allocation errors such as memory leaks.

Chaperon also detects memory blocks that have been allocated and not freed. Such a block is "in use." If a block is in use and cannot be reached by starting from the stack, or statically allocated regions, and proceeding through already reached allocated blocks, then the block is a "memory leak." The block could not be freed without some pointer to specify its address as the parameter to `free()`. At `exit()` Chaperon reports memory leaks automatically.

When Chaperon detects improper behavior, it issues an error message identifying the kind of error and the place where it occurred. Improper behavior is considered to be any access to a logically unallocated region, a Read (or Modify) access to bytes that have been allocated but not yet Written, and errors or abuses of the malloc/free protocol, such as attempting to free the same block twice.

2.2 Source Code Instrumentation

When a more in-depth analysis of the code is required, use the Source Code Instrumentation mode. This mode provides comprehensive memory and error checking through the creation of an instrumented executable of the program. Utilizing the techniques of Source Code Instrumentation (SCI) (patent #5,581,696) and Runtime Pointer Tracking (RPT) (patent # 5,842,019) to develop a comprehensive knowledge of the software, Insure++ builds a database of all program elements, including data structures, memory usage, pointer usage, and interfaces.

Source Code Instrumentation parses, analyzes, and converts the original source code into a new, equivalent source code. The equivalent code is stored in a temporary file that is passed to the compiler, which generates the object code. Throughout the process, the original source code file is not modified and the entire procedure does not require any user intervention. Once all of the files in the project are instrumented, they are linked into a final executable, which is then ready for runtime error detection.

During compilation Insure++ inserts test and analysis functions around every line of source code. At runtime Insure++ checks each data value and memory reference against its database to verify consistency and correctness. Because Source Code Instrumentation allows Insure++ to get deep inside the application under test, it is significantly more precise and thorough than object-level technologies.

2.2.1 Mutation Testing

The Source Code Instrumentation mode also uses Mutation Testing, which is the process of rewriting source code to flush out ambiguities that can cause errors, such as bad copy constructors. Insure++ reads the source code and writes out new source code that is functionally equivalent to the original code but also contains error checking code.

During the error-detection process, the “functionally equivalent” mutants are run in place of the original source code. If the original program does not contain problems, the mutants should not perform any differently than the original program. A mutant that performs differently than the original code indicates a serious error in the original program. Through this method, Insure++ is able to uncover ambiguities that are difficult to detect with any other method or tool. This is particularly important in C++. For example, Mutation Testing can detect the following types of errors:

- Lack of copy constructors or bad copy constructors.
- Missing or incorrect constructors.
- Wrong order of initialization of code.
- Problems with operations of pointers.

2.2.2 Runtime Pointer Tracking

This technology checks every read and write to memory against an accumulated database of pointers and blocks. As memory management commands such as `malloc`, `new`, `delete`, and `free` are executed, Insure++ updates the memory usage database. This allows Insure++ to track memory accesses in all memory segments with incredible precision. Because Insure++ monitors all pointers and memory blocks in the program, it can detect the instruction which overwrites the last pointer to a memory block. As a result, developers can tell when, and at which line of code, the leak occurred.

2.3 Additional Features

2.3.1 Coverage Analysis with TCA

The Total Coverage Analysis (TCA) add-on works hand-in-hand with Insure++. It lets developers get “beneath the hood” of the program to see which parts are actually tested and how often each block is executed. In conjunction with Insure++, this can improve the efficiency of testing and help developers shorten the time required to deliver more reliable programs.

Coverage analysis information is automatically built in whenever a project is instrumented with Insure++. TCA groups code into logical blocks, where a block is a group of statements that must always be executed as a group. For example, the following code has three statements, but only one block.

```
i = 3;
j = i+3;
return i+j;
```

Because TCA reports coverage by blocks instead of lines, developers don’t need to analyze as much data and they can actually see which paths and statements have been executed.

For more information, please refer to our white paper, Maximizing Test-Suite Coverage: The TCA Solution.

2.3.2 Memory Optimization with Inuse

It can be difficult to fully understand the implications of dynamically allocating memory blocks in a program. Inuse is a graphical utility that allows developers to watch how a program handles memory in real-time. Inuse allows developers to:

- Look for memory leaks.
- See how much memory an application uses in response to particular user events.
- Check the memory usage of an application to see if it matches expectations.
- Look for memory fragmentation to see if different allocation strategies might improve performance.
- Analyze memory usage by function, call stack, and block size.
- Verify that the program works as intended.

For more information, please refer to our white paper, Avoiding Dynamic Memory Problems: A New Solution for Developers.

2.3.3 Threads

Insure++ can “run” multi-threaded applications and detect errors in the threads. Insure++ is able to instrument all threads, track all processes in the application, quickly pinpoint problems, and report specific information on errors found. Insure++ allows developers of multi-threaded applications to rapidly find and fix bugs that would otherwise remain hidden in threads and cause the application to perform incorrectly, or fail to perform at all.

For more information, please refer to our white paper, Threads++: A New Solution to the Multithreading Dilemma.

3. How to Use Insure++

Insure++ can be used during the development process to detect errors and prevent them from causing even bigger problems near the end of the project. The earlier Insure++ is used in the development process the better. You can incorporate Insure++ into the nightly builds that automatically build the application every night. The minimal nightly build should pull all necessary code from the source code repository, clean and compile

that code, then build the application. The ideal nightly build should also run all available test cases (both unit test suites and application test suites, including Insure++) and report any failures that occur. At least, this process minimizes the overhead involved in assembling the application pieces. At best, it ensures that the application continues to run as expected and detects any errors introduced by newly integrated code.

If Insure++ cannot be used on a daily basis, it can also be used in another way. For example, say there is an error that is causing a program to crash. The following procedure provides a plan of attack for finding the crash-causing error.

1. Start with Chaperon mode and run Insure++ on the non-instrumented executable.
2. If any errors are reported, fix them.

   If no errors are reported in Chaperon mode, start using Source Code Instrumentation to instrument each file, one by one, where the error might be located.
3. Keep extending instrumentation file by file until the entire application has been covered, or until the errors are found.

The example below illustrates how more errors are uncovered in an application as the level of error detection is increased.

Note: In this section, the error messages are taken from the Windows NT version of Insure++ as displayed in the Insra window. The Insra window is also available in the UNIX versions.

3.1 Using Chaperon

For situations that require a quick overview of the code, Chaperon gives developers a fast, accurate analysis while uncovering extremely complex errors such as memory leaks, memory reference errors, and memory corruption.

For example, say we have a program that is a simple text editor called `editor_demo`. The program appears to run perfectly well when we exercise some of its functions (i.e. opening a new file, clicking the **Help** icon, etc.). But when the existing executable file is run with Insure++ in Chaperon mode, two errors are reported in two of its source files, as shown in Figure 1. On Windows this is accomplished by opening the Insra window, choosing **File> Run**, and selecting the executable. On Linux, this is accomplished by creating an executable and running it with the command `chaperon <program name>`.

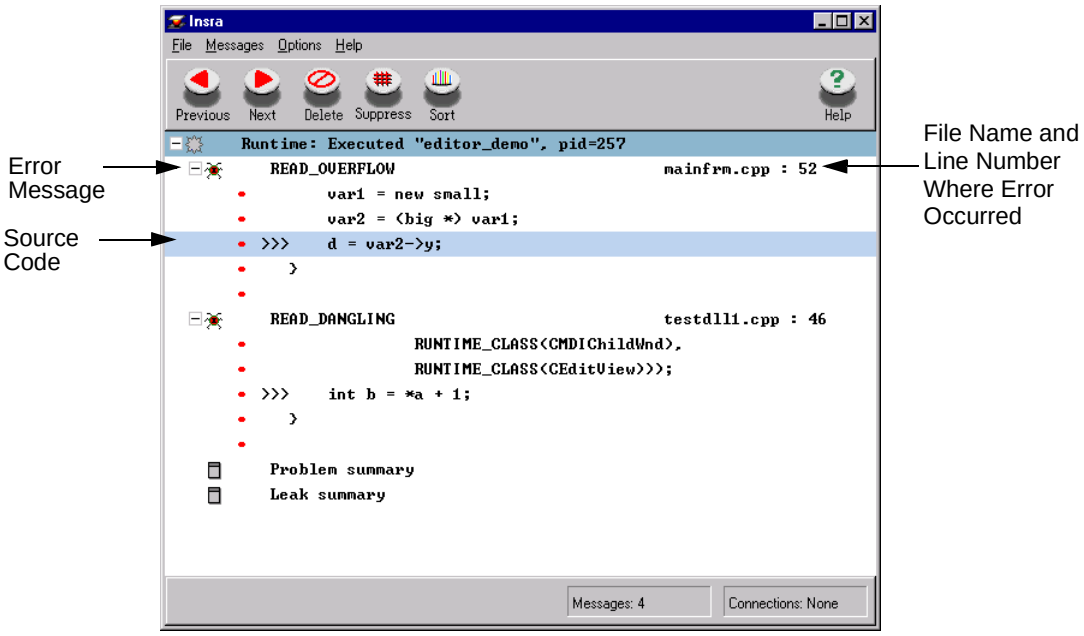


Figure 1. Insure++ error report in Insra for `editor_demo` (Chaperon)

The messages are displayed in the Insra error display GUI. The READ_OVERFLOW message refers to a structured reference that is out of range. The READ_DANGLING message refers to an attempt to perform a read from a dangling pointer. Double-clicking the error message opens a message window (shown in Figure 2) that displays more detailed information about the error message, as well as the stack trace.

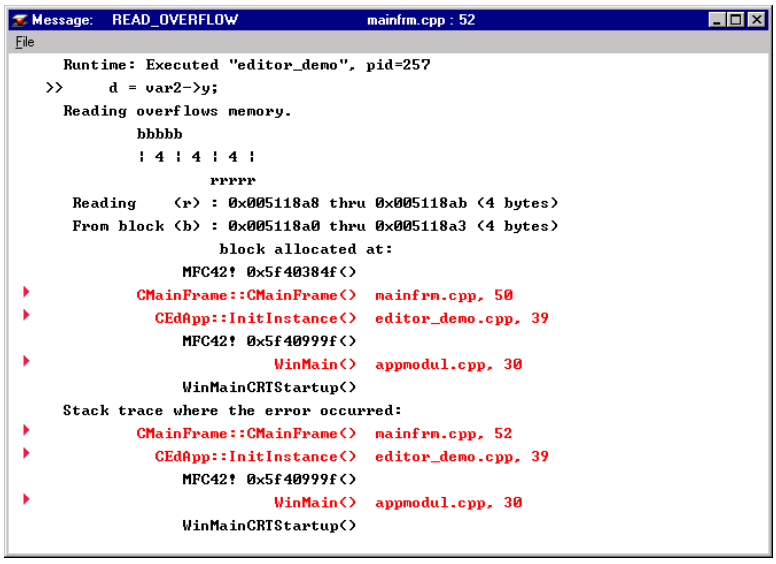


Figure 2. Error message window for READ_OVERFLOW

3.2 Using Source Code Instrumentation

In the previous section Chaperon was used on an executable. Chaperon provides a very effective mode of error detection, but is not quite as thorough as Source Code Instrumentation. Though Source Code Instrumentation is more extensive in its testing functions than Chaperon, it can be much slower.

To minimize the time necessary for testing, developers do not have to instrument the entire application with Insure++; they can instrument one file at a time. Using the same example we used in the section for Chaperon (`editor_demo`), we select and instrument the file `funcs.cpp` (shown in Figure 3). On UNIX this is accomplished by substituting the command `insure` for your compiler. In Developer Studio on Windows, this is accomplished by selecting the file and clicking a button in the Insure++ tool bar.

```
#include <stdlib.h>

int alloc1() {
        char *a;
        a = new char;
        free(a);
        return 0;
}

int delmis1() {
        int *a = new int [5];
        delete a;
        return 0;
}
```

Figure 3. `funcs.cpp`

When the application is recompiled, relinked, and run with Insure++, two new errors are reported by Insure++ in the Insra display window, as shown in Figure 4.

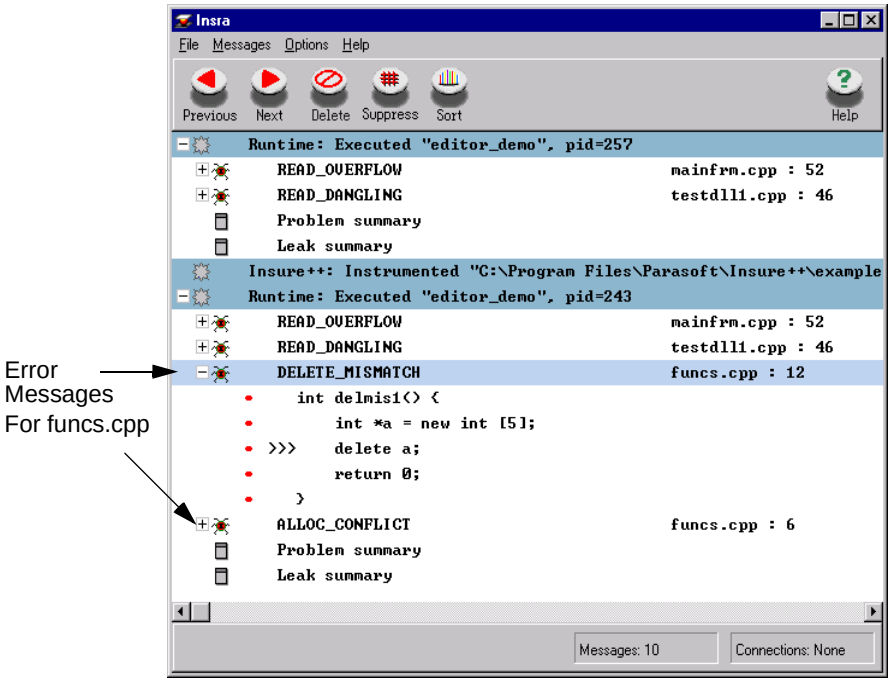


Figure 4. Insure++ error report in Insra

The errors found in `funcs.cpp` are for a memory allocation conflict and the inconsistent use of a `delete` operator. When the entire application is instrumented and run with Insure++, memory leaks are detected as shown in Figure 5.

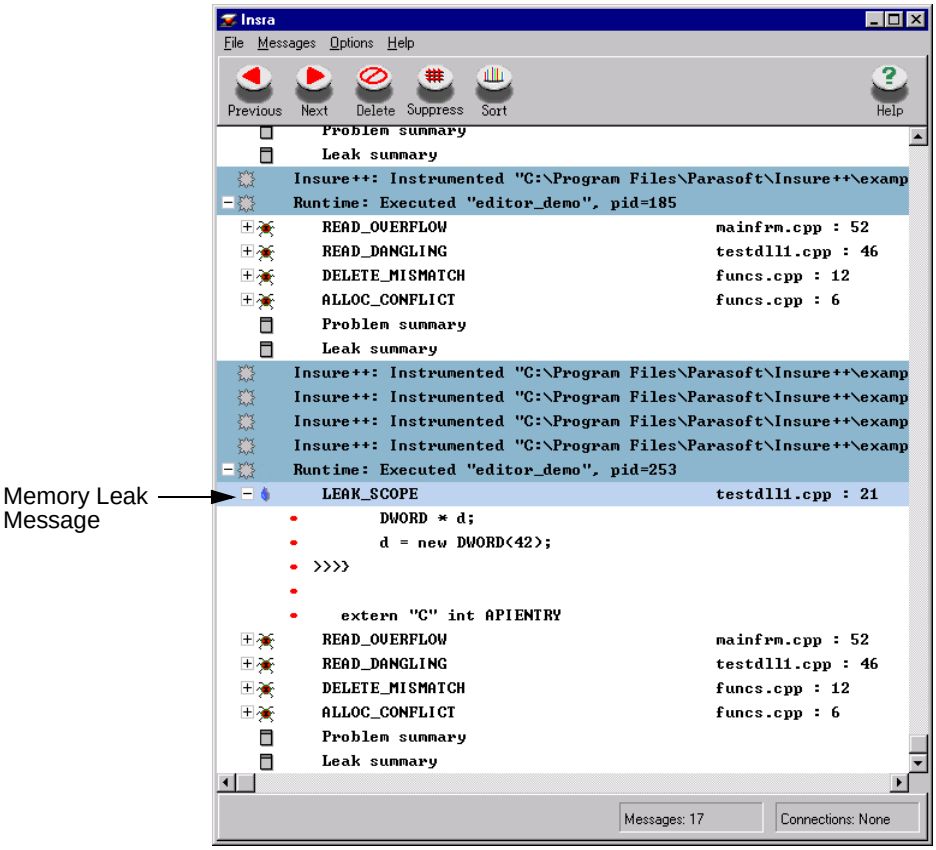


Figure 5. Insure++ error report for `editor_demo` (full instrumentation)

This additional error is for memory leak leaving scope. A function contains a local variable that points to a block of memory but the function returns without saving the pointer in a global variable or passing it back to its caller.

4. What Insure++ Finds

Insure++ finds elusive types of errors in C/C++ applications. Most of these errors fall into three major categories:

- **Memory Reference Errors**: Including memory corruption, pointer abuse, memory leaks, dynamic memory manipulation, and strings.
- **Programming and Third-Party Library Interface Errors**: Including data representation and variable declaration problems, I/O statements, mismatched arguments, invalid parameters in system calls, and unexpected errors in system calls.
- **C++-Specific Errors**: Including problems with inconsistent usage of delete operators and memory allocation conflicts.

Note: The error messages displayed in this section are taken from the standard UNIX version of Insure++.

4.1 Memory Reference Errors

Insure++ checks all types of memory references, including those to static (global), stack, and shared memory, as well as dynamically allocated memory. Even programs that compile, produce correct results, and have a large commercial distribution can contain memory reference errors. These errors are land mines in the memory space, lying dormant until a program accesses the erroneous memory location(s), causing the program to execute incorrectly or crash.

4.1.1 Memory Corruption

These errors can be particularly unpleasant, especially if they are well-disguised. For example, the program shown in Figure 6 concatenates the arguments given on the

command line and prints the resulting string. When compiled with a typical compiler and run, the expected results appear.

```
1:      /*
2:       * File: hello.c
3:       */
4:      main(argc, argv)
5:              int argc;
6:              char *argv[];
7:      {
8:              char str[16];
9:              int i;
10:
11:             str[0] = '\0';
12:             for(i=0; i<argc; i++) {
13:                     strcat(str, argv[i]);
14:                     if(i < (argc-1)) strcat(str, " ");
15:             }
16:             printf("You entered: %s\n", str);
17:      }
```

Figure 6. `hello` **program with bug**

If Figure 7 were the extent of your test procedures, you would probably conclude that this program works correctly. However, it has a very serious memory corruption error.

```
$ cc -o hello hello.c
$ hello world
You entered: hello world
$ hello cruel world
You entered: hello cruel world
```

Figure 7. Test of the `hello` **program**

If you compile the program with Insure++, the command "`hello cruel world`" generates the errors shown in Figure 8. The string that is being concatenated becomes longer than the 16 characters allocated in the declaration at line 8.

```
[hello.c:13] **WRITE_OVERFLOW**
>> strcat(str, argv[i]);

        Writing overflows memory: str

                bbbbbbbbbbbbbbbbbbbbbbbbbb
                |          16               |   2   |
                WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW

        Writing (w) : 0xf7fff8a8 thru 0xf7fff8b9
                                (18 bytes)
        To block (b)    : 0xf7fff8a8 thru 0xf7fff8b7
                                (16 bytes)
                                str, declared at hello.c, 8

                        main() hello.c, 13

**Memory corrupted. Program may crash!!**

[hello.c:16] **READ_OVERFLOW**
>> printf("You entered: %s\n", str);

        String is not null terminated within range: str

        Reading         : 0xf7fff8a8 thru 0xf7fff8b9 (18 bytes)
        From block      : 0xf7fff8a8 thru 0xf7fff8b7 (16 bytes)
                                str, declared at hello.c, 8

                        main() hello.c, 16

You entered: hello cruel world
```

Figure 8. Insure++ error report for the `hello` program

Insure++ finds problems related to overwriting memory or reading past the legal bounds of an object, regardless of whether it is allocated statically (through a global variable), locally on the stack, dynamically (with `malloc`), or even as a shared memory block. It also detects when a pointer crosses from one block of memory into another and starts to overwrite memory there, even if the memory blocks are adjacent.

4.1.2 Pointer Abuse

Problems with pointers are among the most difficult encountered by C and C++ programmers. Insure++ detects pointer-related problems in the following categories:

- Operations on `NULL` pointers.
- Operations on uninitialized pointers.
- Operations on pointers that don't actually point to valid data.

- Operations which try to compare or otherwise relate pointers that don't point at the same data object.
- Attempts to make function calls through function pointers which don't actually point to functions.

Figure 9 shows the code for a second version of the "`hello`" program that uses dynamic memory allocation. When this program is compiled and run with Insure++, an "uninitialized pointer" error at line 22 is reported, because the first time through the argument loop the variable `string_so_far` has not been set to anything.

```
1       /*
2:       * File: hello.c
3:       */
4:      #include <malloc.h>
5:
6:      main(argc, argv)
7:              int argc;
8:              char *argv[];
9:      {
10:             char *string, *string_so_far;
11:             int i, length;
12:
13:             length = 1;             /* Include last NULL */
14:
15:             for(i=0; i<argc; i++) {
16:                     length += strlen(argv[i])+1;
17:                     string = malloc(length);
18:     /*
19:      * Copy the string built so far.
20:      */
21:                     if(string_so_far != (char *)0)
22:                             strcpy(string, string_so_far);
23:                     else *string = '\0';
24:
25:                     strcat(string, argv[i]);
26:                     if(i < argc-1) strcat(string, " ");
27:                     string_so_far = string;
28:             }
29:             printf("You entered: %s\n", string);
```

Figure 9. `hello` **program with dynamic memory allocation**

4.1.3 Memory Leaks

Memory leaks can be extremely difficult to detect because they can take days of continuous execution to cause a failure. Small leaks in low level routines can mean that the function might be called thousands or even millions of times before it loses enough memory to crash the system. This is exactly the type of subtle bug that survives in-house

testing only to show up when a customer uses the program for some enormous processing task.

A memory leak occurs when a piece of dynamically allocated memory can no longer be freed because the program no longer contains any pointers to that block. A simple example of this behavior can be seen by running the (corrected) "`hello`" program with the arguments:

```
hello this is a test
```

If we examine the state of the program just prior to execution of line 27 (Figure 9), we find:

- The variable `string_so_far` points to the string "`hello`", which it was assigned as a result of the previous loop iteration.
- The variable `string` points to the extended string "`hello this`", which was assigned on this loop iteration.

These assignments are shown schematically in Figure 10 - both variables point to different blocks of dynamically allocated memory.

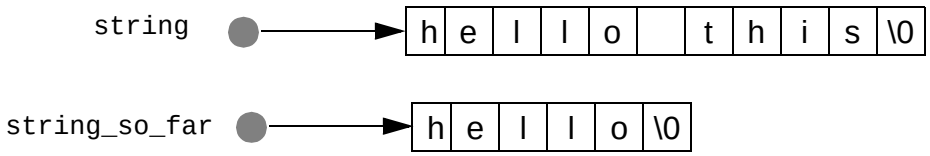


Figure 10. Pointer assignments before the memory leak

But when line 27 is executed,

```
string_so_far = string;
```

both variables are made to point to the longer memory block, as shown in Figure 11.

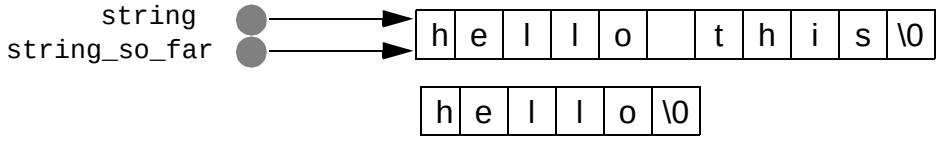


Figure 11. Pointer assignments after the memory leak

Once this has happened, there is no remaining pointer that points to the shorter block. There is now no way that the memory that was previously pointed to by `string_so_far` can be reclaimed; it is permanently allocated. When this program is compiled and run with Insure++, a “memory leak” error is reported, as shown in Figure 12.

```
[hello.c:27] **LEAK_ASSIGN**
>>              string_so_far = string;

        Memory leaked due to reassignment: string

Lost block:     0x0001fbb0 thru 0x0001fbb6 (7 bytes)
                        string, allocated at hello.c, 17
        Stack trace where the error occurred:
                                main() hello.c, 27
```

Figure 12. Insure++ error report for memory leak in `hello` program

Since this error is caused when a pointer is re-assigned, it is known as `LEAK_ASSIGN` in Insure++. Other types of memory leaks detected by Insure++ include:

- `LEAK_FREE`: Occurs when you free a block of memory that contains pointers to other memory blocks. If no other pointers point to these secondary blocks, they are permanently lost and will be reported by Insure++.
- `LEAK_RETURN`: Occurs when a function returns a pointer to an allocated block of memory but the returned value is ignored in the calling routine.
- `LEAK_SCOPE`: Occurs when a function contains a local variable that points to a block of memory but the function returns without saving the pointer in a global variable or passing it back to its caller.

4.1.4 Dynamic Memory Manipulation

With dynamically allocated memory, programs often continue running after a programming error corrupts the memory; sometimes they don’t crash at all. One common mistake is trying to reuse a pointer after it has already been `freed`. This “dangling pointer” problem often goes unnoticed, because many machines and compilers allow this particular behavior.

In addition to dangling pointers, Insure++ detects many other dynamic memory bugs including:

- Freeing the same memory block multiple times.
- Attempting to delete or free statically allocated memory.
- Freeing stack memory (local variables).
- Passing a pointer to `delete` or `free` that doesn't point to the start of a memory block.
- Calls to `delete` or `free` with `NULL` or uninitialized pointers.
- Passing nonsensical arguments or arguments of the wrong data type to `malloc`, `calloc`, `realloc` or `free`.
- Mismatch between calls to `new []` and `delete []`.
- Mixing calls between `malloc`, `new`, `free`, and `delete`. For example, obtaining a pointer with `malloc` and freeing it with `delete`.
- Problems with overloading `new` and `delete` operators.

Insure++ also helps uncover dynamic memory problems through its `RETURN_FAILURE` error code. For example, Insure++ does not normally issue an error message if `malloc`, returns a `NULL` pointer because it is out of memory. This behavior is the default because it is assumed that the user program is already checking for and handling this case. However, if your program appears to be failing for this reason, you can enable the `RETURN_FAILURE` error message class. Insure++ will then print a message whenever a system call fails.

The program in Figure 13 has an error in calling operator `delete` at line 34. The `new` operator at line 29 allocated an array of class objects `first`. The array is deallocated with the `delete` operator. However, the `delete` operator is not called with `[]`, so it doesn't call the destructor for the store class for each element of the array `first`. This type of behavior can lead to serious problems in C++ programs.

```
 1: #include <iostream.h>
 2: #include <stdlib.h>
 3: #include <string.h>
 4: #define SIZE 128
 5:
 6: class store{
 7: public:
 8:             store(int sz=24){size = sz;}
 9:             ~store(){size=0; cout << “Deleting:” << ptr ;}
10:             int copy(char *src){
11:                     int len;
12:
13:                     len = strlen(src);
14:                     if(len > size)return(0);
15:                     strcpy(ptr,src);
16:                     return(len);
17:             }
18: protected:
19:             char ptr[SIZE];
20:             int size;
21: };
22:
23: main()
24: {
25: class store *first;
26: int limit=5,i;
27: char local[128];
28:
29:             first = new store[limit];
30:             for(i=0;i<limit;i++){
31:                     sprintf(local,”Element %d\n”,i);
32:                     first[i].copy(local);
33:             }
34:             delete first;
35: }
```

Figure 13. Program with error in `delete`

When this program is compiled and run with Insure++, a “delete mismatch” error is reported, as shown in Figure 14.

```
[bracket.C:33] **DELETE_MISMATCH**
>> delete first;
Inconsistent usage of delete operator: first
array deleted without []
Stack trace where the error occurred:
                                        main() bracket.C, 33
```

Figure 14. Insure++ error report for `delete[]`

4.1.5 Strings

Though the standard C library string handling functions are useful, unfortunately they are also a rich source of potential errors. They do very little checking on the bounds of the objects being manipulated. Insure++ detects most string-related problems such as overwriting the end of a buffer and using strings without null terminators.

4.2 Programming and Third-Party Library Interface Errors

Like memory reference errors, programming errors act as land mines in the program. When executed, programs containing them encounter problems ranging from incorrect values to system crashes. Third-party errors are particularly tricky to handle. When a program crashes in the third-party code, it might be because either the library was passed incorrect parameters by the user or there is an actual bug in the third-party code. Programming errors, and library interface errors in particular, can also cause difficulties porting code across platforms due to architectural idiosyncrasies.

Insure++ is able to discover both of these types of problems and provide diagnostics about which of the problems occurred. Providing this type of information is critical because source code for the third-party library is generally not available, and it is not enough just to say that the bug occurred in the library. If the bug in the third-party library was caused by passing incorrect parameters, you can easily remedy it. But if the bug is caused by a fault in the library, then all that can be done is report it to the vendor.

Insure++ can discover bugs in the third-party library even if the source code to the library is not available. The bugs are discovered by the code that Insure++ links to the application. When the third-party library executes, the Insure++ library checks the code in the third-party library for memory overwrites, dynamic memory problems, and more.

4.2.1 Data Representation and Variable Declaration Problems

Many programs make either explicit or implicit assumptions about the various data types on which they operate. On many workstations, pointers and integers have the same number of bytes. While some of these problems can be detected during compilation, some codes go to great lengths to hide these problems with typecasts such as:

```
char *p;
int ip;

ip = (int)p;
```

On many systems, this type of operation is valid and causes no problems. However, problems can arise when this code is ported to other architectures. For example, the code shown above will fail when executed on a PC (16-bit integer, 32-bit pointer) or a 64-bit architecture such as DEC TruUnix64 (32-bit integer, 64-bit pointer). In cases where the

operation loses information, Insure++ will report an error. On machines for which the data types have the same number of bits, no error is reported.

Insure++ also detects inconsistent declarations of variables between source files. For example, an object might be declared as an array in one file, but as a pointer in another. Insure++ reports size differences so that an array declared as one size in one file and another in a second will be detected.

4.2.2 I/O Statements

The `printf` and `scanf` family of functions are easy places to make mistakes which show up either as bugs or portability problems. For instance, data input into a variable with data type `double` will be incorrect when the format specified in the call to `scanf` is different (for example, `float`).

In addition to checking `printf` and `scanf` arguments, Insure++ also detects errors in other I/O statements. The code

```
foo(line)
        char line[80];
{
        gets(line);
}
```

works as long as the input supplied by the user is shorter than 80 characters, but fails on longer input. Insure++ checks for this case and reports an error if necessary.

4.2.3 Mismatched Arguments

Calling functions with incorrect arguments is a common and often overlooked problem in many programs. Insure++ automatically detects the argument mismatches described below.

- **Sign errors**: Arguments agree in type but one is signed and the other unsigned, e.g., `int` vs. `unsigned int`.
- **Compatible types**: The arguments are different data types which happen to occupy the same amount of memory on the current machine, e.g. `int` vs. `long`, if both are thirty-two bits. While this error may not cause problems on your current machine, it is a portability problem.
- **Incompatible types**: Data types are fundamentally different or require different amounts of memory. `int` vs. `long` would appear in this category on machines where they require different numbers of bits.
- **Alias errors**: If you use `typedef` to define new names for data types, Insure++ generates an error when you use them inconsistently.

4.2.4 Invalid Parameters in System Calls

Interfacing to library software is often tricky because passing an incorrect argument to a routine might cause it to fail inconsistently. Debugging such problems is much harder than correcting your own code since you typically have much less information about how the library routine should work. Insure++ has built-in knowledge of a large number of system calls and checks the arguments to ensure correct data type and, if appropriate, range. For example, the code

```
myrewind(fp)
        FILE *fp;
{
        fseek(fp, (long)0, 3);
}
```

will cause an error, since the last argument passed to the `fseek` function is outside the legal range.

Insure++ includes built-in interface tests for hundreds of libraries. You can construct additional interface checks easily with Insure++'s interface definition features.

4.2.5 Unexpected Errors in System Calls

Checking return codes from system calls and dealing correctly with all the possible error cases that can arise is very difficult. Exhaustive testing of all the possible combinations is almost impossible. As a result, programs can fail unexpectedly because some system call fails in a way that had not been anticipated. The consequences can range from a "core dump" to an infrequent, unrepeatable error.

Insure++ has a special error class, `RETURN_FAILURE`, that detects these problems. All system calls known to Insure++ contain special error checking code that detects failures. Normally, these errors are suppressed since it is assumed that the application is handling them itself, but they can be enabled at runtime. With `RETURN_FAILURE` enabled Insure++ detects errors in system calls. It prints the routine name, arguments, and an error description for the following errors:

- `malloc` runs out of memory.
- Files which don't exist.
- Incorrectly set permission flags.
- Incorrect use of I/O routines.
- Exceeding the limit on open files.
- Interprocess communication and shared memory errors.
- Unexpected "interrupted system call" errors.

Insure++ understands standard UNIX and Windows system calls, the X Window System, Motif, and many other popular libraries.

4.3 Errors Specific to C++

Insure++ uncovers a number of errors specific to the C++ language, including memory allocation conflicts and inconsistent usage of the delete operator.

4.3.1 Inconsistent Usage of Delete Operator

The current version of ANSI C++ distinguishes between memory allocated with `new` and `new[]`. A `delete` call *must* (according to the standard) match the `new` call, i.e. whether or not it has `[]`. Calling `new[]` and `delete` might cause the compiler to not call the destructor on each element of the array, which can lead to serious errors. Even worse, if the memory was allocated differently, memory may be corrupted. This is definitely poor practice and unlikely to work with future releases of the specific compiler.

The code in Figure 15 shows a block of memory allocated with `new[]` and freed with `delete`, without `[]`.

```
1:      /*
2:       * File: delmis1.cpp
3:       */
4:
5:      int main() {
6:              int *a = new int [5];
7:              delete a;
8:              return 0;
9:      }
```

Figure 15. Program `delmis1.cpp`

When this program is compiled and run with Insure++, a “delete mismatch” error is reported, as show in Figure 16.

```
[delmis1.cpp:7] **DELETE_MISMATCH**
>>              delete a;

        Inconsistent usage of delete operator: a

        array deleted without []
                        a, allocated at:
                                main()  delmis1.cpp, 6

        Stack trace where the error occurred:
                                main()  delmis1.cpp, 7
```

Figure 16. Insure++ error report for `delmis1.cpp`

4.3.2 Memory Allocation

This error is generated when a memory block is allocated with `new (malloc)` and freed with `free (delete)`. Insure++ distinguishes between two possible causes of this problem.

- `badfree`: Memory was allocated with `new` or `new[]` and an attempt was made to free it with `free`.
- `baddelete`: Memory was allocated with `malloc` and an attempt was made to free it with `delete` or `delete[]`.

Some compilers allow this, but it is not a good programming practice and might cause a problem with portability.

The code in Figure 17 shows a typical example of this error, allocating a block of memory with `malloc` and then freeing it with `delete`.

```
1:      /*
2:       * File: alloc2.cpp
3:       */
4:      #include <stdlib.h>
5:
6:      int main() {
7:              char *a;
8:
9:              a = (char *) malloc(1);
10:             delete a;
11:             return 0;
12:     }
```

Figure 17. Program with error in allocating memory

When this program is compiled and run with Insure++, a “memory allocation conflict” error is reported, as show in Figure 18.

```
[alloc2.cpp:10] **ALLOC_CONFLICT**
>>              delete a;

      Memory allocation conflict: a

      delete operator used to deallocate memory not
          allocated by new
              block allocated at:
                  malloc()(interface)
                    main()alloc2.cpp, 9

      Stack trace where the error occurred:
                    main()alloc2.cpp, 10
```

Figure 18. Insure++ error report for `alloc2.cpp`

5. Conclusion

Even programs that compile, produce correct results, and have a large commercial distribution can contain elusive errors such as memory references and memory leaks. Insure++ can detect these errors during development and prevent them from holding up a project, or appearing at a user site.

6. Availability

Insure++ is available now at http://www.parasoft.com. To learn more about how Insure++ and other ParaSoft development tools can help your department prevent and detect errors, talk to a Software Quality Specialist today at 1-888-305-0041 (U.S.A. only), or visit http://www.parasoft.com.

6.1 Platforms

Insure++ is available for the following platforms (64-bit support is available on selected platforms).

- Windows NT/2000
- UNIX, including: DEC (Alpha 4.x, TruUnix64 5), HP (HPUX 10.x & 11.x), IBM (AIX 4.3.x), Linux (glibc 2.2.4 or higher), SGI (Irix 6.5), Solaris/Sparc (7, 8), and Solaris x86 (7, 8)

6.2 Compiler Compatibility

Insure++ works with all popular compilers.

- Windows: integrates into Visual C++ 6.0.
- UNIX: compilers include `CC`, `gcc`, and `HP C++`.

Insure++ can also check modules written in languages other than C++, such as Fortran, Ada, Pascal, etc.

7. Contacting ParaSoft

USA

2031 S. Myrtle Ave.
Monrovia, CA 91016
Tel: (888) 305-0041 (toll-free)
(626) 305-0041
Fax: (626) 305-3036
E-mail: info@parasoft.com
URL: www.parasoft.com

Europe

France: Tel: (33 1) 64 89 26 00
UK: Tel: +44 (020) 8263 2827
Germany: Tel: (49) 7805/9569 -60
Mail: info-europe@parasoft.com