PARASOFT®

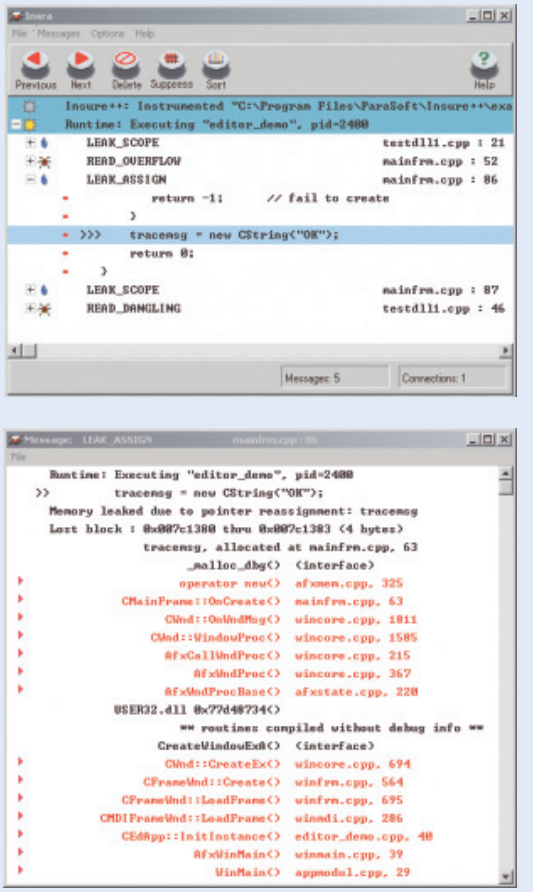
PARASOFT Insure++®

Insure++ is a runtime memory analysis and error detection tool for C and C++ that automatically identifies a variety of difficult-to-track programming and memory-access errors, along with potential defects and inefficiencies in memory usage. Errors such as memory corruption, memory leaks, access outside of array bounds, invalid pointers, and the like often go undetected during normal testing, only to result in application crashes in the field. Insure++ will help you find and eliminate such defects in your applications to ensure the integrity of their memory usage.

Errors Detected

During testing, Insure++ checks all types of memory references, including those to static (global), stack, and shared memory — both in user's code and in third party libraries. Errors that Insure++ detects include:

- Corrupted heap and stack memory
- Use of uninitialized variables and objects
- Array and string bounds errors on heap and stack
- Use of dangling, NULL, and uninitialized pointers
- All types of memory allocation and free errors or mismatches
- All types of memory leaks
- Type mismatches in global declarations, pointers, and function calls
- Some varieties of dead code (compile-time)



On Windows and UNIX operating systems, you can send error messages to Insra, the error display GUI, and then click to see full error explanations and stack trace information.

Multiple Use Modes

Insure++ has highly detailed memory analysis capabilities that are based on patented* source instrumentation algorithms. Source code instrumentation enables Insure++ to detect more error types than other memory error detection technologies, and also provides complete information indicating the root causes of the errors found, using a full database of program elements and memory structures.

There are two ways to use Insure++ for memory analysis and error detection. The first and most detailed analysis is achieved with **full source code instrumentation**. This requires that application sources be compiled and linked with Insure++, which generates its own instrumented files that are passed to the actual compiler.

The second option is **linking with Insure++**, which provides a trade-off between the extent of error reporting and the actual time to build and run an instrumented application. In this mode, Insure++ can detect and report most of the error types, including leaks, bad memory references, standard API usage errors, and so on.

Features

- Detection of memory corruption on heap and stack
- Detection of uninitialized variables, pointers, and objects
- Detection of memory leaks and other memory allocation/free errors
- STL checking** for proper usage of STL containers and related memory errors
- Compile-time checks for type- and size-related errors
- Runtime tracing of function calls
- GUI and command line interface
- Memory error checking in 3rd party static and dynamic libraries
- Direct interfaces with Visual Studio debugger

Benefits

- Finds memory errors before they become runtime problems
- Finds common errors during 64 bit porting
- Helps optimize memory usage of applications
- Reduces development and support costs
- Easily integrates with regression test suites in "smoke alarm" mode
- Provides detailed stack traces of errors to help understand their causes

TCA Test Coverage

- Calculates line and block coverage
- Reports line, block, class, function, and file coverage
- Text reports and interactive code browser with coverage highlighting

Inuse Memory Monitor

- Visualizes memory leaks
- Displays memory use in real time
- Helps correlate memory usage with program events

** Available for any Unix users with GCC 3.0 and above.

Parasoft® Inuse® and Parasoft® TCA®

Along with the runtime memory error detection engine, Insure++ includes two components that increase the tool's scope of analysis:

- TCA (provides total coverage analysis)
- Inuse (provides application memory usage analysis)

TCA analyzes and reports block code coverage and lets you get "beneath the hood" of your program to see which parts are actually tested and how often each block is executed. In conjunction with a runtime error detection tool like Insure++ and a comprehensive test suite, this can dramatically improve the efficiency of your testing and promote faster delivery of more reliable programs.

Inuse visualizes how an application uses memory. This component provides a graphical view of all memory allocations, over time, with specific visibility into overall heap usage, block allocations, possible outstanding leaks, and so on. By providing insight into an application's memory usage patterns, Inuse allows you to effectively analyze and optimize runtime memory usage and performance.

* Parasoft holds patents #5,581,696 and #6,085,029 for its source instrumentation algorithms.

Supported Platforms

Microsoft Windows (32-bit, 64-bit)
- Visual C++

Linux 32 and 64 bit
- GNU gcc/g++
- Intel ICC

Solaris UltraSparc Processor
- Forte Developer
- Sun Studio
- GNU gcc/g++

IBM AIX, PowerPC 32 and 64 bit processor
- IBM Visual Age
- IBM Visual Age (xIC compilers)
- GNU gcc/g++

PARASOFT®

BITTT

BITTT Enterprises, Inc. Increases Code Quality, Stability, and Compliance with Parasoft Development Testing Platform

BITTT Enterprises, Inc. specializes in business processes and provides strategic business solutions for information management. BITTT helps their clients improve internal technology systems, increasing efficiency and productivity for a healthier bottom line.

Timothy W. Okrey, Managing Partner, is in charge of development at BITTT. In fact, he is the mastermind behind the code that BITTT writes. Recently, Okrey was continuing development on an on-going project that had been in the works for a couple of years. The program was in virtual production when it suddenly started crashing. The situation left Okrey completely dumbfounded.

After trying to resolve the problem on his own and hitting brick walls in every direction, Okrey discovered Parasoft's development testing solution for runtime analysis and error detection. Parasoft's Development Testing Platform not only helped Okrey resolve the issue at hand, but also enabled him to simultaneously and effectively enhance a dozen separate projects.

Emerging of a Critical Error

The product that Okrey was developing had been stable and running in a virtual production mode. But the program started failing after a recent build to address a number of enhancements requested by the customer.

BITTT had invested two years on the product, a payroll-related solution designed to help the customer bring down the 60 to 70 man hours invested every week to manually complete payroll for 1000 employees in 14 states. As a result of BITTT's work, their customer's payroll was now automated, enabling them to spend less than 12 man hours on it each week. Unfortunately, the show-stopping error that emerged with the latest build caused BITTT's customer to revert back to their manual payroll process.

Based on 20+ years of development experience, Okrey knows that if you run into a brick wall, it's time to redo the entire project a different way. Unfortunately, that wasn't even an option in this situation because "There was no smoking gun or even a traceable error." Okrey explains further, "This was not new development. Nor did we try to pull parts of code together to make it work. This particular program was written from scratch using a toolkit as the backend for the details."

The toolkit is one that Okrey started creating in 1993. It allows him to pull working functions into raw source code or use them as a library for any project. The toolkit provides a stable foundation for all of his projects and alleviates the need to rewrite code over and over again. This toolkit has grown to well over 500,000 lines of code, which was written with utmost diligence. Okrey strictly follows the rules of structured programming and is judicious about keeping his code clean. He had never used a third party tool to analyze his code and has never had the need.

Reducing the Length and Cost of Downstream Development Processes

For over a week, Okrey tried to re-engineer different pieces of the class that was causing trouble. But his attempts at fixing the problem only resulted in changing some of the internals so that the point of failure occurred in a different location. "I spent more than 40 hours going through all of my code with a fine tooth comb and a magnifying glass, like I usually do. I was unable to locate the problem. I could see what was happening; I just could not see why it was happening," Okrey said.

That's when his search for help began. He found only a handful of tools that were able to do what he wanted. Of that handful, most of the products merely allowed for static review of code. Parasoft was the only product that also performed dynamic analysis. "Parasoft gives me the ability to analyze my content in the environment where it's being run as opposed to just looking at the code on paper," Okrey said.

After getting set up and running, Parasoft ran through the first build of Okrey's source code—all 500,000+ lines. Within 15 seconds of launching, Parasoft surfaced a stale pointer error. "If I hadn't found Parasoft, it would have led to very drastic requirements from the client," Okrey said as he reflected on the rapid return on investment. He added that "To go from a functioning version of the program to a non-functioning version simply due to an upgrade would have led to a reversal of progress and forced financial concessions that I do not want to even consider. It was an ugly situation."

Increasing Code Quality, Stability, and Compliance

Parasoft enabled Okrey to completely revamp the toolkit source code; specifically, improving string-handling. The improvement spread to other projects. Okrey has dozens of other programs for various clients that use the same backend toolkit, so all of these programs reaped the benefits. Okrey states, "I can't even begin to tell you all of the programs that are dependent on the backend toolkit. As a result of the improvements Parasoft enabled me to make, they are all that much more stable and compliant."

Okrey said that the Parasoft Development Testing Platform gave him the ability to implement and enforce his high coding standards. "Parasoft forces you to verify that the standards and practices that are being used are absolutely pristine," Okrey said. "One of the challenges as a project leader—or managing partner, like myself—is confirming that your team is writing code that meets high standards. Parasoft can help me verify that my team is writing code that meets my standards and allow me to guarantee results. I am really excited about that."

Finding Value in Parasoft's Development Testing Platform

Okrey is pleased with the quality that Parasoft has rapidly ingrained into his application development process. Not only has he been able to rectify a problem for a valued customer, but he has also been able to improve the quality of dozens of programs for other customers.

Okrey says, "I'm very particular about products that I choose to endorse. The majority of software written in the world just doesn't work the way it's supposed to work for various reasons. Maybe it's poorly designed so it runs slow, or system requirements aren't realistic. The list goes on.

"However, there are a few products that I really like. One of those is a system software product that I've come to rely on. I've never experienced a GPF with it. Never. When I learned that the provider of that product was a Parasoft customer, that was it. That's what made me decide to give Parasoft a try and I'm very happy that I did."

PARASOFT

USA PARASOFT HEADQUARTERS
101 E. Huntington Drive, Monrovia, CA 91016
Phone: (888) 305-0041, Email: info@parasoft.com

PARASOFT

Automating C/C++ Runtime Error Detection with Parasoft Insure++

C and C++ developers have a unique problem: many errors in their code don't manifest themselves during testing. Software with subtle problems such as memory corruption may run flawlessly on one machine, but crash on another. To help developers find and fix such problems prior to release, Parasoft designed Parasoft Insure++.

Parasoft Insure++ is an automated runtime application testing tool that detects elusive errors such as memory corruption, memory leaks, memory allocation errors, variable initialization errors, variable definition conflicts, pointer errors, library errors, I/O errors, and logic errors. With the click of a button or a simple command, Insure++ automatically uncovers the defects in your code— helping you to identify the source of that strange problem you've been trying to diagnose for weeks, as well as alerting you to problems that you were previously unaware of. Insure++ detects more errors than any other tool because its patented technologies achieve the deepest possible understanding of the code under test and expose even the most elusive problems.

This paper discusses the challenges associated with C and C++ development— including memory corruption, memory leaks, pointer errors, I/O errors, and more— and explains how Insure++ helps you eliminate those problems.

What Problems Can Insure++ Find?

Insure++ automatically detects errors that might otherwise go unnoticed in normal testing. Subtle memory corruption errors and dynamic memory problems often don't crash the program or cause it to give incorrect answers until the program is delivered to customers and they run it on *their* systems... and then the problems start. Even if Insure++ doesn't find any problems in your programs, running it gives you the confidence that your program doesn't contain any errors.

Of course, Insure++ can't possibly check everything that your program does. However, its checking is extensive and covers every class of programming error, including:

- Memory corruption due to reading or writing beyond the valid areas of global, local, shared, and dynamically allocated objects.
- Operations on uninitialized, NULL, or "wild" pointers.
- Memory leaks.
- Errors allocating and freeing dynamic memory.
- String manipulation errors.
- Operations on pointers to unrelated data blocks.
- Invalid pointer operations.
- Incompatible variable declarations.
- Mismatched variable types in `printf` and `scanf` argument lists.

The following sections detail the types of errors that Insure++ detects.

Memory Corruption

This is one of the most unpleasant errors that can occur, especially if it is well disguised. As an example of what can happen, consider the program shown below. This program concatenates the arguments given on the command line and prints the resulting string:

```
/*
 * File: hello.c
 */
#include <string.h>

main(argc, argv)
```

```
    int argc;
    char *argv[];
{
    int i;
    char str[16];

    str[0] = '\0';
    for(i=0; i<argc; i++) {
        strcat(str, argv[i]);
        if(i < (argc-1)) strcat(str, " ");
    }
    printf("You entered: %s\n", str);
    return (0);
}
```

If you compile and run this program with your normal compiler, you'll probably see nothing interesting. For example:

```
c:\source> cl /Zi hello.c
         c:\source> hello
         You entered: hello
         c:\source>hello world
         You entered: hello world
         c:\source>hello cruel world
         You entered: hello cruel world
```

If this were the extent of your test procedures, you would probably conclude that this program works correctly, despite the fact that it has a very serious memory corruption bug.

If you compile with Insure++, the command `hello cruel world` generates the errors shown below, because the string that is being concatenated becomes longer than the 16 characters allocated in the declaration at line 7:

```
[hello.c:15] **WRITE_OVERFLOW**
>>         strcat(str, argv[i]);

  Writing overflows memory: <argument 1>

          bbbbbbbbbbbbbbbbbbbbbbbbb
          |           16           | 2 |
          wwwwwwwwwwwwwwwwwwwwwwwwwwwww

   Writing  (w) : 0xbfffeed0 thru 0xbfffeee1 (18 bytes)
   To block (b) : 0xbfffeed0 thru 0xbfffeedf (16 bytes)
                  str, declared at hello.c, 11

  Stack trace where the error occurred:
                          strcat()  (interface)
                            main()  hello.c, 15

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

[hello.c:18] **READ_OVERFLOW**
>>     printf("You entered: %s\n", str);

  String is not null terminated within range: str

  Reading   : 0xbfffeed0
  From block: 0xbfffeed0 thru 0xbfffeedf (16 bytes)
              str, declared at hello.c, 11
```

```
  Stack trace where the error occurred:
                            main()  hello.c, 18

You entered: hello cruel world
```

Insure++ finds all problems related to overwriting memory or reading past the legal bounds of an object, regardless of whether it is allocated statically (that is, a global variable), locally on the stack, dynamically (with `malloc` or `new`), or even as a shared memory block.

Insure++ also detects situations where a pointer crosses from one block of memory into another and starts to overwrite memory there, even if the memory blocks are adjacent.

Pointer Abuse

Problems with pointers are among the most difficult encountered by 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.
- Function calls through function pointers that don't actually point to functions.

Below is the code for a second attempt at the "Hello world" program that uses dynamic memory allocation:

```
/*
 * File: hello2.c
 */
#include <stdlib.h>
#include <string.h>

main(argc, argv)
    int argc;
    char *argv[];
{
    char *string, *string_so_far;
    int i, length;

    length = 0;

    for(i=0; i<argc; i++) {
        length += strlen(argv[i])+1;
        string = malloc(length+1);

/*
 * Copy the string built so far.
 */
        if(string_so_far != (char *)0)
            strcpy(string, string_so_far);
        else *string = '\0';

        strcat(string, argv[i]);
        if(i < argc-1) strcat(string, " ");
        string_so_far = string;
    }
    printf("You entered: %s\n", string_so_far);
    return (0);
```

```
}
```

The basic idea of this program is that we keep track of the current string size in the variable `length`. As each new argument is processed, we add its length to the `length` variable and allocate a block of memory of the new size. Notice that the code is careful to include the final `NULL` character when computing the string length (line 11) and also the space between strings (line 14). Both of these are easy mistakes to make. It's an interesting exercise to see how quickly Insure++ finds such an error.

The code in lines 19-24 either copies the argument to the buffer or appends it, depending on whether or not this is the first pass round the loop. Finally, in line 25, we point at the new longer string by assigning the pointer `string` to the variable `string_so_far`.

If you compile and run this program under Insure++, you'll see "uninitialized pointer" errors reported for lines 19 and 20. This is because the variable `string_so_far` hasn't been set to anything before the first trip through the argument loop.

Memory Leaks

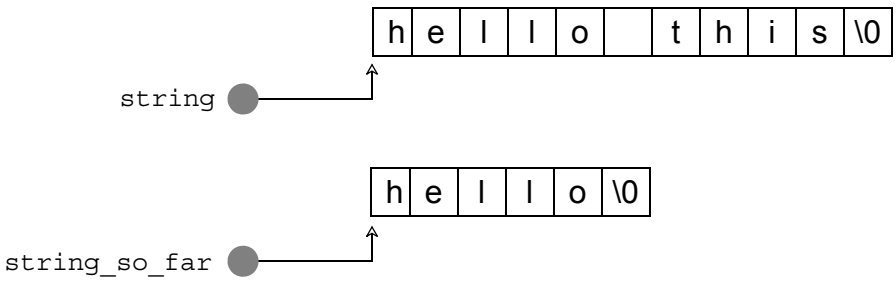
A "memory leak" occurs when a piece of dynamically allocated memory cannot be freed because the program no longer contains any pointers that point to the block. A simple example of this behavior can be seen by running the (corrected) "Hello world" program with the arguments

```
hello3 this is a test
```

If we examine the state of the program at line 27, just before executing the call to `malloc` for the second time, we observe:

- 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 below; both variables point to blocks of dynamically allocated memory.

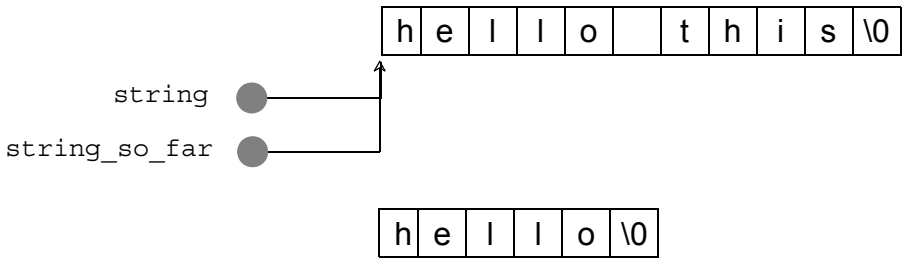


Pointer assignments before the memory leak

The next statement

```
string_so_far = string;
```

will make both variables point to the longer memory block as shown below.



Pointer assignments after the memory leak

Once this happens, however, there is no remaining pointer that points to the shorter block. Even if you wanted to, there is no way that the memory that was previously pointed to by `string_so_far` can be reclaimed; it is permanently allocated. This is known as a “memory leak” and is diagnosed by Insure++ as shown below:

```
[hello3.c:28] **LEAK_ASSIGN**
>>          string_so_far = string;

  Memory leaked due to pointer reassignment: string

  Lost block : 0x0804bd68 thru 0x0804bd6f (8 bytes)
               string, allocated at hello3.c, 18
                         malloc()  (interface)
                           main()  hello3.c, 18

  Stack trace where the error occurred:
                           main()  hello3.c, 28
```

This example is called `LEAK_ASSIGN` by Insure++ since it is caused when a pointer is re-assigned. Other types of leaks that Insure++ detects include:

Table 1:

Leak Type	Description
`LEAK_FREE`	Occurs when you free a block of memory that contains pointers to other memory blocks. If there are no other pointers that point to these secondary blocks, then 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.

Notice that Insure++ indicates the exact source line on which the problem occurs, which is a key issue in finding and fixing memory leaks. This is an extremely important feature because it’s easy to introduce subtle memory leaks into your applications, but very hard to find them all. Using Insure++, you can instantly pinpoint the line of source code which caused the leak.

Dynamic Memory Manipulation

Using dynamically allocated memory properly is another tricky issue. In many cases, programs continue running well after a programming error causes serious memory corruption; sometimes they don't crash at all.

One common mistake is to try to reuse a pointer after it has already been freed. As an example, we could modify the "Hello world" program to de-allocate memory blocks before allocating the larger ones. Consider the following piece of code which does just that:

```
22:      if(string_so_far != (char *)0) {
23:              free(string_so_far);
24:              strcpy(string, string_so_far);
25:      }
26:      else *string = '\0';
```

If you run this code (`hello4.c`) through Insure++, you'll get another error message about a "dangling pointer" at line 23. The term "dangling pointer" is used to mean a pointer that doesn't point at a valid memory block anymore. In this case, the block is freed at line 22 and then used in the following line. This is another common problem that often goes unnoticed because many machines and compilers allow this particular behavior.

In addition to this error, Insure++ also detects the following errors:

- Reading from or writing to "dangling pointers."
- Passing "dangling pointers" as arguments to functions or returning them from functions.
- Freeing the same memory block multiple times.
- Attempting to free statically allocated memory.
- Freeing stack memory (local variables).
- Passing a pointer to `free` that doesn't point to the beginning of a memory block.
- Calls to `free` with `NULL` or uninitialized pointers.
- Passing non-sensical arguments or arguments of the wrong data type to `malloc`, `calloc`, `realloc` or `free`.

Another way that Insure++ can help you track down dynamic memory problems is through the `RETURN_FAILURE` error code. Normally, Insure++ will not issue an error 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.

If your program appears to be failing due to an unchecked return code, you can enable the `RETURN_FAILURE` error message class. Insure++ will then print a message whenever any system call fails.

Strings

The standard C library string handling functions are a rich source of potential errors because they do very little checking on the bounds of the objects being manipulated.

Insure++ detects problems such as overwriting the end of a buffer as described in "Memory Corruption" on page 1. Another common problem is caused by trying to work with strings that are not null-terminated, as in the following example:

```
/*
 * File: readovr2.c
 */
main()
{
```

```
    char junk;
    char b[8], c[8];
    strncpy(b, "This is a test",
                        sizeof(b));
    memset(c, 0, sizeof(c));
    printf("%s\n", b);
    return (0);
}
```

This program attempts to copy the string `This is a test` into a buffer which is only 8 characters long. Although it uses `strncpy` to avoid overwriting its buffer, the resulting copy doesn't have a `NULL` on the end. Insure++ detects this problem in line 10 when the call to `printf` tries to print the string.

Uninitialized Memory

A particularly unpleasant problem to track down occurs when your program makes use of an uninitialized variable. These problems are often intermittent and can be particularly difficult to find using conventional means, since any alteration in the operation of the program may result in different behavior. It is not unusual for this type of bug to show up and then immediately disappear whenever you attempt to trace it.

Insure++ performs checking for uninitialized data in two sub-categories.

Table 2:

Category	Name	Description
1.	`copy`	Normally, Insure++ doesn't complain when you assign a variable using an uninitialized value because many applications do this without error. In many cases, the value is changed to something correct before being used, or may never be used at all.
2.	`read`	Insure++ generates an error report whenever you use an uninitialized variable in a context which cannot be correct, such as an expression evaluation.

To clarify the difference between these categories, consider the following code:

```
1:      /*
2:       * File: readuni1.c
3:       */
4:      #include <stdio.h>
5:
6:      int main()
7:      {
8:              struct rectangle {
9:                      int width;
10:                     int height;
11:             };
12:
13:             struct rectangle box;
14:             int area;
15:
16:             box.width = 5;
17:             area = box.width*box.height;
18:             printf("area = %d\n", area);
19:             return (0);
20:     }
```

In line 17, the value of `box.height` is used to calculate a value which is invalid because its value was never assigned. Insure++ detects this error in the `READ_UNINIT_MEM(read)` category. This category is enabled by default, so a message will be displayed.

If you changed line 17 to

```
17:             area = box.height;
```

Insure++ would report errors of type `READ_UNINIT_MEM(copy)` for both lines 17 and 18, but only if you had unsuppressed this error category.

Unused Variables

Insure++ can also detect variables that have no effect on the behavior of your application, either because they are never used, or because they are assigned values that are never used. In most cases, these are not serious errors because the offending statements can simply be removed, and so they are suppressed by default.

Occasionally, however, an unused variable may be a symptom of a logical program error, so you may wish to enable this checking periodically.

Data Representation Problems

Many programs make either explicit or implicit assumptions about the various data types on which they operate. A common assumption made on workstations is that pointers and integers have the same number of bytes. While some of these problems can be detected during compilation, others hide operations with typecasts, such as shown in the following example:

```
char *p;
int ip;

ip = (int)p;
```

On many systems, this type of operation would be valid and would not cause any problems. However, problems can arise when such code is ported to alternative architectures. The code shown above would fail, for example, when executed on a PC (16-bit integer, 32-bit pointer) or a 64-bit architecture such as the Compaq Tru64 Unix (32-bit integer, 64-bit pointer).

In cases where such an operation loses information, Insure++ reports an error. On machines for which the data types have the same number of bits (or more), no error is reported.

Incompatible Variable Declarations

Insure++ detects inconsistent declarations of variables between source files. A common problem is caused when an object is declared as an array in one file (for example, `int myblock[128];`), but as a pointer in another (for example, `extern int *myblock;`).

See the files `baddecl1.c` and `baddecl2.c` in the `examples` directory for an example. Insure++ also reports differences in size, so that an array declared as one size in one file and a different size in another will be detected.

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 example, consider the following code:

```
foo()
{
        double f;

        scanf("%f", &f);
```

```
}
```

This code will not crash, but the value read into the variable `f` will not be correct, since its data type (`double`) doesn't match the format specified in the call to `scanf` (`float`). As a result, incorrect data will be transferred to the program.

In a similar way, the example `badform2.c` corrupts memory, since too much data will be written over the supplied variable:

```
foo()
{
        float f;

        scanf("%lf", &f);
}
```

This error can be very difficult to detect.

A more subtle issue arises when data types used in I/O statements "accidentally" match. The following code functions correctly on machines where types `int` and `long` have the same number of bits, but fails otherwise:

```
foo()
{
        long l = 123;
        printf("l = %d\n", l);
}
```

Insure++ detects this error, but classifies it differently from the previous cases. You can choose to ignore this type of problem while still seeing the previous bugs.

In addition to checking `printf` and `scanf` arguments, Insure++ also detects errors in other I/O statements. The following code works as long as the input supplied by the user is shorter than 80 characters, but it fails on longer input:

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

Insure++ checks for this case and reports an error if necessary.

Note: This case is somewhat tricky, since Insure++ can only check for an overflow after the data has been read. In extreme cases, the act of reading the data will crash the program before Insure++ gets the chance to report it.

Mismatched Arguments

Calling functions with incorrect arguments is a common problem in many programs, and can often go unnoticed. For example, Insure++ detects the error in the following program in which the argument passed to the function `foo` in `main` is an integer rather than a floating point number:

```
double foo(dd)
        double dd;
{
        return dd + 1.0;
}

main()
{
        printf("Result = %f\n", foo(1));
}
```

Note: Converting this program to ANSI style (for example, with a function prototype for `foo`) makes it correct since the argument passed in `main` will be automatically converted to `double`. Insure++ doesn't report an error in this case.

Insure++ detects several different categories of errors, which you can enable or suppress separately depending on which types of bugs you consider important.

- Sign errors: Arguments agree in type, but one is signed and the other unsigned (for example, `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 (for example, `int` vs. `long` if both are 32-bits). While this error might not cause problems on your current machine, it is a portability problem.
- Incompatible types: Similar to the example above. 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.

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 in an unpredictable manner. 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 you pass to ensure correct data type and, if appropriate, correct range.

For example, the following code would generate an error since the last argument passed to the `fseek` function is outside the legal range:

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

Unexpected Errors In System Calls

Checking the return codes from system calls and dealing correctly with all the error cases that can arise is a very difficult task. Very rarely will a program deal with all possible cases correctly.

An unfortunate consequence of this is that programs can fail unexpectedly because some system call fails in a way that had not been anticipated. The consequences of this can range from a nasty "core dump" to a system that performs erratically at the customer location.

Insure++ has a special error class, `RETURN_FAILURE`, that can be used to detect these problems. All the system calls known to Insure++ contain special error checking code that detects failures. These errors are normally suppressed, since it is assumed that the application is handling them itself, but they can be enabled at runtime by unsuppressing the reporting of `RETURN_FAILURE` errors. Any system call that returns an error code will then print a message indicating the name of the routine, the arguments supplied, and the reason for the error.

This capability detects *any* error in *any* known system call. Among the potential benefits is automatic detection of errors in the following situations:

- `malloc` runs out of memory.
- Files that do not exist.
- Incorrectly set permission flags.
- Incorrect use of I/O routines.
- Exceeding the limit on open files.

- Inter-process communication and shared memory errors.
- Unexpected “interrupted system call” errors.

How Does Insure++ Work?

Insure++ performs static analysis at compile-time as well as sophisticated dynamic analysis at runtime. Using a unique set of patented technologies, Insure++ develops a comprehensive knowledge of the software and all of its elements under test. During compilation, Insure++ reads and analyzes the source code, then inserts test and analysis functions around every line of source code. Insure++ builds a database of all program elements, and then at runtime, Insure++ checks each data value and memory reference against its database to verify consistency and correctness. For a quick test, re-link your application against Insure++’s runtime library and run the program as you normally would. For a deeper look, instrument your code to zoom in on errors and perform the most thorough testing possible.

Insure++ checks all types of memory references, including those to static (global), stack, and shared memory. It can find memory corruption and memory leaks as well as errors allocating and freeing dynamic memory. Insure++ also checks third-party libraries and functions. Testing with Insure++, you can automatically find such errors as string manipulation errors, operations on uninitialized pointers, operations on pointers to unrelated data blocks, invalid pointer operations, incompatible variable declarations, and mismatched variable types.

To help you optimize dynamic memory usage and maximize test suite coverage, Insure++ includes two complementary tools: Parasoft Inuse and Parasoft TCA. Inuse is a graphical utility that lets you watch a program allocate and free dynamic memory blocks, helping you understand the memory usage patterns of algorithms and optimize their behavior. TCA shows test coverage analysis for an application by determining how many files, functions, and statements have been executed, giving you an idea of the overall quality of testing.

The following sections provide a more detailed look at key Insure++ technologies and features.

Source Code Instrumentation and Runtime Pointer Tracking

Patented Source Code Instrumentation (patents #5,581,696 and #6,085,029) and Runtime Pointer Tracking (patent # 5,842,019) technologies allow Insure++ to develop a comprehensive knowledge of the software and all of its elements under test. During compilation, Insure++ reads and analyzes the source code, then inserts test and analysis functions around every line of source code. Insure++ builds a database of all program elements, and then at runtime, Insure++ checks each data value and memory reference against its database to verify consistency and correctness. This allows Insure++ to track memory accesses with incredible precision and in all memory segments (including heap, static, and stack memory), and is especially critical for detection of memory leaks. 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 what line of code, the leak occurred.

Source Code Instrumentation provides more thorough error detection than Object Code Instrumentation (OCI). OCI-based tools read the object code generated by compilers, and before programs are linked, they are instrumented. The basic principle of these tools is that they look for processor instructions that access memory. In the object code, any instruction that accesses memory is modified to check for corruption. Because these tools are triggered by memory instructions, they can only detect errors related to memory. These tools can detect errors in dynamic memory, but they have limited detection ability on the stack and they do not work on static memory. They cannot detect any other type of errors because of the weaknesses in OCI technology. At the object level, a lot of significant information about the source code is permanently lost and cannot be used to help locate errors. Another drawback of these tools is that they cannot detect when memory leaks occur. Pointers and integers are not distinguishable at the object level, making the cause of the leak undetectable.

Mutation Testing

Insure++ leverages techniques from traditional Mutation Testing to uncover ambiguities that are difficult to detect through other methods or tools. Whereas traditional Mutation Testing attempts to create "faulty" mutants to create a more effective test suite, Insure++ creates and executes what should be functionally equivalent mutants of the source code under test. When one of these mutants performs differently than the original program, it indicates that the code's functionality relies on implicit assumptions which may not always be satisfied during execution. If a mutant causes the program to crash or encounter other serious problems, it's a sign that when the assumptions are not satisfied, serious errors will occur at runtime.

Because Mutation Testing can uncover a number of otherwise hard-to-find or esoteric errors, it is particularly important in C++, where there are so many opportunities to make errors. 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.
- Dependence on undefined behavior such as order of evaluation.

TCA

The Total Coverage Analysis (TCA) tool works hand-in-hand with Insure++ to show you which parts of code you've tested and which you've missed. With TCA, you can stop wasting time testing the same parts of code over and over again and start exercising untested code instead. TCA shows test coverage analysis for an application by determining how many files, functions, and statements have been executed, giving you an idea of the overall quality of testing.

TCA provides the following coverage information:

- **Overall Summary:** Shows the percentage covered at the application level (i.e., summed over all program entities).
- **Function Summary:** Displays the coverage of each individual function in the application.
- **Block Summary:** Displays the coverage broken down by individual program statement blocks.

Unlike some other coverage analysis tools which work only on a line-by-line basis, TCA is able to group your code into logical blocks. A block is a group of statements which must always be executed as a group. Advantages of using blocks instead of lines include:

- Lines of code which have several blocks are treated separately.
- Grouping equivalent statements into a single block reduces the amount of data you need to analyze.
- By treating labels as a separate group, you can actually detect which paths have been executed in addition to which statements.

For more details on TCA, see the Parasoft white paper "Maximizing C/C++ Test Suite Coverage."

Inuse

Inuse is a graphical "memory visualization" tool that helps you determine where unseen leaks and other memory abuses may be hurting your program. Inuse allows you to watch how your program allocates and frees dynamic memory blocks, in real-time. Inuse will help you to better understand the memory usage patterns of algorithms and how to optimize their behavior. With Inuse, you'll have a clear understanding of how your program actually uses (and abuses) memory.

You can use Inuse to:

- See how much memory an application uses in response to particular user events.
- Compare an application's overall memory usage to its expected memory usage.
- Detect the most subtle memory leaks, which can cause problems over time.
- Look for memory fragmentation to see if different allocation strategies might improve performance.
- Identify other common memory problems, including memory blowout, memory overuse, and memory bottlenecks.
- Analyze memory usage by function, call stack, and block size.

Inuse provides the following reports to help you achieve these objectives:

- **Heap History**: This graph displays the amount of memory allocated to the heap and the user process as a function of real (that is, wall clock) time. This display updates periodically to show the current status of the application, and can be used to keep track of the application over the course of its execution.
- **Block Frequency**: This graph displays a histogram showing the number of blocks of each size that have been allocated. It is useful for selecting potential optimizations in memory allocation strategies.
- **Heap Layout:** This graph shows the layout of memory in the dynamically allocated blocks, including the free spaces between them. You can use this report to “see” fragmentation and memory leaks. Clicking any block in the heap layout will tell you the block’s address, size, and status (free, allocated, overhead, or leaked). Clicking an allocated or leaked block will also open a window telling you the block id, block address, stack size, and stack trace for the selected block.
- **Time Layout**: This graph shows the sequence of allocated blocks. As each block is allocated, it is added to the end of the display. As blocks are freed, they are marked green. From this display, you can see the relative size of blocks allocated over time. For example, this will allow you to determine if you are allocating a huge block at the beginning of the program or many small blocks throughout the run.
- **Usage Summary**: This bar graph shows how many times each of the memory manipulation calls has been made. It also shows the current size of the heap and the amount of memory actively in use. (The heap fragmentation can be computed simply from these numbers as `(total-in_use)/total`).
- **Usage Comparison**: Graphically compares memory from different runs of one executable or among runs of different executables.
- **Query:** The query function enables you to “view” blocks of memory allocated by your program according to their id numbers, their size, and/or their stack traces. You can edit the range of the query according to block id, block size, and stack trace. By “grouping” blocks of memory in this way, you can better understand how memory is being used in your program. The range options let you narrow or broaden your query to your specifications. For example, you can see how much memory is being allocated from a single stack trace or by the entire program combined. For each query you can choose whether you receive a detailed (i.e. containing block id, block size, and stack trace information) or summarized report.

For more details on Inuse, see the Parasoft white paper "Avoiding C/C++ Dynamic Memory Problems."

Integrating Insure++ Into Your Development Process

Using Insure++ is easy. You simply recompile your program with Insure++ instead of your normal compiler. Running the program under Insure++ then generates a report whenever an error is detected. For each error found, Insure++ reports the name of related variables, the line of source code containing the error, a description of the error, and a stack trace.

Insure++ can benefit all C/C++ projects, regardless of their configuration or scope. 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++.

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, by incorporating Insure++ into the software development process earlier. By integrating Insure++ into your development environment, you can save weeks of debugging time and prevent costly crashes from affecting your customers. You can also use Insure++ with other Parasoft tools to prevent and detect software errors from the design phase all the way through testing and QA.

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.

Insure++ can be used throughout the software development lifecycle— from exposing problems during development, to diagnosing problems in released/deployed applications. It can be used in concert with Parasoft C++Test to improve C/C++ code reliability, functionality, security, performance, and maintainability. Moreover, Insure++ works as part of a comprehensive team-wide Automated Error Prevention solution that reduces delivery delays and improves the quality and security of complex, multi-language enterprise applications..

Learning More

Insure++ is available at http://www.parasoft.com. To learn more about how Insure++ and other Parasoft solutions can help your organization prevent errors, contact Parasoft today or visit http://www.parasoft.com.

About Parasoft

For over 25 years, Parasoft has researched and developed software solutions that help organizations define and deliver defect-free software efficiently. By integrating Development Testing, API/Cloud Testing, and Service Virtualization, we reduce the time, effort, and cost of delivering secure, reliable, and compliant software. Parasoft's enterprise and embedded development solutions are the industry's most comprehensive—including static analysis, unit testing with requirements traceability, functional & load testing, dev/test environment management, and more. The majority of Fortune 500 companies rely on Parasoft in order to produce top-quality software consistently and efficiently.

Contacting Parasoft

USA	Phone: (888) 305-0041	Email: info@parasoft.com
GERMANY	Phone: +49 731 880309-0	Email: info-de@parasoft.com
POLAND	Phone: +48 12 290 91 01	Email: info-pl@parasoft.com
UK	Phone: +44 (0)208 263 6005	Email: sales@parasoft-uk.com
FRANCE	Phone: (33 1) 64 89 26 00	Email: sales@parasoft-fr.com
ITALY	Phone: (+39) 06 96 03 86 74	Email: c.soulat@parasoft-fr.com
NORDICS	Phone: +31-70-3922000	Email: info@parasoft.nl
OTHER	See http://www.parasoft.com/contacts	