



# **User's Guide**

**Version 6.0**

**UNIX**

**ParaSoft Corporation  
2031 S. Myrtle Ave.  
Monrovia, CA 91016  
Phone: (888) 305-0041  
Fax: (626) 305-9048  
E-mail: [info@parasoft.com](mailto:info@parasoft.com)  
URL: [www.parasoft.com](http://www.parasoft.com)**

## IMPORTANT NOTICE

**LIMITED USE OF SOFTWARE LICENSE.** This agreement contains the ParaSoft Corporation (PARASOFT) Limited Use Software License Agreement (AGREEMENT) which will govern the use of the ParaSoft products contained within it.

YOU AGREE TO THE TERMS OF THIS AGREEMENT BY THE ACT OF OPENING THE ENVELOPE CONTAINING THE SOFTWARE OR INSTALLING IT ON YOUR COMPUTER SYSTEM. DO NOT OPEN THE ENVELOPE OR ATTEMPT TO INSTALL THE SOFTWARE WITHOUT FIRST READING, UNDERSTANDING AND AGREEING TO THE TERMS AND CONDITIONS OF THIS AGREEMENT. YOU MAY RETURN THIS PRODUCT TO PARASOFT FOR A FULL REFUND BEFORE OPENING THE ENVELOPE OR INSTALLING THE SOFTWARE.

**GRANT OF LICENSE.** PARASOFT hereby grants you, and you accept, a limited license to use the enclosed electronic media, user manuals, and any related materials (collectively called the SOFTWARE in this AGREEMENT). You may install the SOFTWARE in only one location on a single disk or in one location on the temporary or permanent replacement of this disk. If you wish to install the SOFTWARE in multiple locations, you must either license an additional copy of the SOFTWARE from PARASOFT or request a multi-user license from PARASOFT. You may not transfer or sub-license, either temporarily or permanently, your right to use the SOFTWARE under this AGREEMENT without the prior written consent of PARASOFT.

**DERIVED PRODUCTS.** Products developed from the use of the SOFTWARE remain your property. No royalty fees or runtime licenses are required on said products.

**TERM.** This AGREEMENT is effective from the day you open the sealed package containing the electronic media and continues until you return the original SOFTWARE to PARASOFT, in which case you must also certify in writing that you have destroyed any archival copies you may have recorded on any memory system or magnetic, electronic, or optical media and likewise any copies of the written materials.

**PARASOFT'S RIGHTS.** You acknowledge that the SOFTWARE is the sole and exclusive property of PARASOFT. By accepting this agreement you do not become the owner of the SOFTWARE, but you do have the right to use the SOFTWARE in accordance with this AGREEMENT. You agree to use your best efforts and all reasonable steps to protect the SOFTWARE from use, reproduction, or distribution, except as authorized by this AGREEMENT. You agree not to disassemble, de-compile or otherwise reverse engineer the SOFTWARE.

**YOUR ORIGINAL ELECTRONIC MEDIA/ARCHIVAL COPIES.** The electronic media enclosed contain an original PARASOFT label. Use the original electronic media to make "back-up" or "archival" copies for the purpose of running the SOFTWARE program. You should not use the original electronic media in your terminal except to create the archival copy. After recording the archival copies, place the original electronic media in a safe place. Other than these archival copies, you agree that no other copies of the SOFTWARE will be made.

**CUSTOMER REGISTRATION.** PARASOFT may from time to time revise or update the SOFTWARE. These revisions will be made generally available at PARASOFT's discretion. Revisions or notification of revisions can only be provided to you if you have signed and returned the enclosed Registration Card to PARASOFT and if

your electronic media are the originals. PARASOFT's customer services are available only to registered users.

**LIMITED WARRANTY.** PARASOFT warrants for a period of thirty (30) days from the date of purchase, that under normal use, the material of the electronic media will not prove defective. If, during the thirty (30) day period, the software media shall prove defective, you may return them to PARASOFT for a replacement without charge. PARASOFT further allows thirty (30) days free consultation regarding the SOFTWARE, its installation and operation. After this initial period PARASOFT reserves the right to refuse further assistance unless a maintenance contract has been purchased.

**SUITABILITY.** PARASOFT has worked hard to make this a quality product, however PARASOFT makes no warranties as to the suitability, accuracy, or operational characteristics of this SOFTWARE. The SOFTWARE is sold on an "as-is" basis.

**EXCLUSIONS.** PARASOFT shall have no obligation to support SOFTWARE that is not the then current release.

**LIMITATION OF LIABILITY.** You agree that PARASOFT's liability for any damages to you or to any other party shall not exceed the license fee paid for the SOFTWARE.

PARASOFT WILL NOT BE RESPONSIBLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OF THE SOFTWARE ARISING OUT OF ANY BREACH OF THE WARRANTY, EVEN IF PARASOFT HAS BEEN ADVISED OF SUCH DAMAGES. THIS PRODUCT IS SOLD "AS-IS".

SOME STATES DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU. YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.

**TERMINATION OF AGREEMENT.** If any of the terms and conditions of this AGREEMENT are broken, PARASOFT has the right to terminate the AGREEMENT and demand that you return the SOFTWARE to PARASOFT. At that time you must also certify, in writing, that you have not retained any copies of the SOFTWARE.

**ENTIRE AGREEMENT.** This AGREEMENT constitutes the entire agreement between you and PARASOFT.

All brand and product names are trademarks or registered trademarks of their respective holders.

Copyright © 1993-2001

ParaSoft Corporation

2031 South Myrtle Avenue, Monrovia, CA 91016

Printed in the U.S.A., September 18, 2001



# Table of Contents

## Introduction

Welcome! .....	1
Insure++ Installation, Startup, and Licensing .....	5
Contacting ParaSoft .....	15

## Getting Started

Running Insure++ .....	17
Preventing Errors With CodeWizard .....	32
Analyzing Code Coverage With TCA .....	33
Optimizing Dynamic Memory With Inuse .....	34

## Using Insure++

Insure++ .....	35
Chaperon (Linux Only) .....	53
Reports .....	67
Insra .....	84
Selective Checking .....	97
Interacting with Debuggers .....	98
Tracing .....	104
Signals .....	107
Code Insertions .....	109
Interfaces .....	113

## Inuse

Introducing Inuse .....	135
Running Inuse .....	140
Inuse Reports .....	145

# TCA

Introducing TCA .....	155
Using TCA.....	157
The TCA Display .....	169
Building a Test Suite .....	173

# Reference

Configuration Options .....	175
Memory Overflow .....	197
Error Codes.....	199
ALLOC_CONFLICT .....	206
BAD_CAST .....	209
BAD_DECL .....	211
BAD_FORMAT .....	214
BAD_INTERFACE .....	219
BAD_PARM .....	221
COPY_BAD_RANGE.....	226
COPY_DANGLING .....	228
COPY_UNINIT_PTR.....	230
COPY_WILD.....	232
DEAD_CODE.....	234
DELETE_MISMATCH .....	237
EXPR_BAD_RANGE .....	240
EXPR_DANGLING .....	242
EXPR_NULL .....	244
EXPR_UNINIT_PTR .....	246
EXPR_UNRELATED_PTRCMP .....	248
EXPR_UNRELATED_PTRDIFF .....	250
EXPR_WILD .....	252
FREE_BODY .....	255
FREE_DANGLING.....	257
FREE_GLOBAL .....	259
FREE_LOCAL.....	261
FREE_UNINIT_PTR .....	263
FREE_WILD .....	265
FUNC_BAD.....	267
FUNC_NULL.....	269
FUNC_UNINIT_PTR .....	271

FUNC_WILD .....	273
INSURE_ERROR.....	275
INSURE_WARNING .....	276
LEAK_ASSIGN .....	278
LEAK_FREE .....	280
LEAK_RETURN.....	283
LEAK_SCOPE .....	285
PARAM_BAD_RANGE.....	287
PARAM_DANGLING .....	289
PARAM_NULL.....	292
PARAM_UNINIT_PTR.....	295
PARAM_WILD.....	297
READ_BAD_INDEX.....	301
READ_DANGLING .....	303
READ_NULL.....	305
READ_OVERFLOW.....	307
READ_UNINIT_MEM.....	313
READ_UNINIT_PTR .....	317
READ_WILD .....	319
RETURN_DANGLING.....	322
RETURN_FAILURE .....	324
RETURN_INCONSISTENT.....	326
UNUSED_VAR.....	328
USER_ERROR .....	331
VIRTUAL_BAD.....	333
WRITE_BAD_INDEX .....	336
WRITE_DANGLING .....	338
WRITE_NULL .....	340
WRITE_OVERFLOW .....	342
WRITE_UNINIT_PTR.....	344
WRITE_WILD.....	346
Insure++ API .....	349
Interface Functions.....	353

## Index

Index .....	359
-------------	-----



# Welcome!

Memory corruption and memory leak errors are extremely difficult to find, often taking weeks (sometimes even months) to track down. Insure++ detects these elusive, crash-causing errors in C/C++ applications. By using a unique set of technologies, including mutation testing, 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. 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 speed up debugging from the design phase all the way through testing and QA.

## New features

Insure++ 6.0 for UNIX contains significant changes from previous versions that will help you debug faster and more efficiently. These include powerful new technologies and refined user interfaces.

New features/enhancements include:

- Chaperon comprehensive memory checking for non-instrumented executables (Linux only)
- More efficient parser
- Improved runtime library, including improved stack trace
- Simplified error reporting configuration

## Pinpointing programming errors

Two of the most serious software-related problems are the time taken to debug (and therefore deliver) a product and the number of bugs that are not detected during testing and which are only found at customer sites.

These problems arise in many different ways. Insure++ finds a wide variety of programming and memory access errors, including:

- Memory corruption due to reading or writing beyond the valid areas of global, local, shared, *and* dynamically allocated objects
- Operations on illegal, or unrelated pointers
- Reading uninitialized memory
- Memory leaks
- Errors allocating and freeing dynamic memory
- String manipulation errors
- Incompatible variable declarations
- Mismatched variable types in `printf` and `scanf` argument lists

Insure++ does not use a "statistical" approach to "trap" memory reference errors, but rather checks each memory reference for validity when that reference is executed. Insure++ checks all types of memory references, including those to static (global) and stack memory, as well as dynamically allocated memory. When Insure++ finds a problem, it provides a complete diagnosis, including the name of related variables, the line of source code containing the error, a description of the error, and a stack trace.

## Checking calls to libraries

Just as it does with memory reference errors, Insure++ finds library interface errors.

- Mismatched argument types or function declarations.
- Invalid parameters in library calls.
- Errors returned by library calls.

Insure++ understands standard UNIX system calls, the X Window system, Motif, and many other popular libraries. On each library call, Insure++ checks that every variable is of the correct type and is within its valid range.

If the source code for a third-party library is available, Insure++ can automatically check it if you rebuild the library with Insure++ as you do your own source code. If you don't have the source code, Insure++ includes utilities which allow you to make a "definition" of their interfaces. Once the interface is completely specified, these libraries will receive the same comprehensive checking that Insure++ provides for standard libraries.

## Debugging technologies

Insure++ is a source-level automatic runtime error detection tool, and you may be wondering why we chose this particular technology instead of one of the other competing debugging technologies. Source Code Instrumentation gets deeper inside the code being checked than other technologies. This allows more complicated and subtle bugs to be detected.

## Code coverage analysis with TCA

The Total Coverage Analysis (TCA) add-on 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.

## Memory optimization with Inuse

You think you know how your program handles memory in real-time, but only Inuse can tell you for sure. Find out where leaks and other memory abuses are hurting your program with this graphical "memory visualization" tool.

## Supported platforms and compilers

The platforms and compilers supported by Insure++ at the time this manual was printed are summarized in the table below:

Welcome!

Introduction

Platform	OS Version	cc	CC	cxx	gcc/g++	x1c/x1C	aCC
AIX	4.3.0.0 4.2.1.0 4.1.3.0	x			x	x	
Alpha	4.0						
HP	11.00 10.20 10.01	x	x		x		x
Linux	2.x				x		
SGI	6.2 6.3 6.4 6.5	x	x		x		
Solaris	2.7 2.6 2.5	x	x		x		

# Insure++ Installation, Startup, and Licensing

## Installation and Startup

The following steps describe the process of installing Insure++ from either

- RPM Package for Linux (PowerTools CD-ROM from RedHat or ftp), or
- an electronic archive obtained via ftp.

Installing Insure++ from a tape or ftp requires no particular system privileges other than the ability to create a directory into which the software will be placed. Installing from a CD-ROM requires root privileges.

The amount of disk space required by Insure++ depends on which system you are installing. The table below shows the approximate size and ARCH label for each supported platform. These labels will help you complete the installation.

System	ARCH	Disk Space (MB)
DEC Alpha Digital UNIX 4.0	alpha4	96
HP 9000 HP-UX 10.X	hp10	93
HP 9000 HP-UX 11.X	hp11	80 (32/64 bit)
Linux	linux	40
Reliant-UNIX 4.1x	sinix	80 (32/64 bit)
AIX 4.X	aix4	145 (64-bit)
SGI IRIX 6.X	sgi6	230 (32/64 bit)

System	ARCH	Disk Space (MB)
Sun Solaris 2.X	solaris	90

**Note:** For SINIX (Reliant-Unix) users, perl is required in order to install Insure++v6.0 for SINIX. If you purchase a CD copy of Insure++, you can find perl packages on your CD. If you don't have perl, you may download a copy from `ftp.mch.sni.de:/sni/mr/pd/perl/perl.5.00404`

Please also note that Insure++6.0 for Linux is only supported on glibc-2.x mode. It is no longer supported on libc.5 (RH4.x) mode.

## Step 1. Create a directory for the Insure++ distribution

Choose a location in which to install Insure++ and make this directory with a command such as

```
mkdir /usr/local/parasoft
```

All subsequent steps must be performed in the new directory, so you should change to it now.

```
cd /usr/local/parasoft
```

**Note:** From now on we will assume that you have chosen to install the software in a directory called `/usr/local/parasoft` as indicated above. If you actually chose a different name, you will have to modify the following commands appropriately.

If you received Insure++ on a tape, proceed to Step 2A. If you received Insure++ on a CD-ROM, skip Step 2A and proceed directly to Step 2B. If you received Insure++ via ftp, skip Step 2 and proceed directly to Step 3.

**NOTE:** To install Insure++ from an RPM Package, follow these steps:

1. Log in as `root`.
2. At the prompt, enter:  
`rpm -ivh (or -uvh) Insure++-6.0-x.i386.rpm`
3. Call 1-888-305-0041 or send an email to `license@parasoft.com` to obtain a license
4. Run `pslic` to add the license.

## Step 2

### Step 2A. Extract the tape contents

Read the tape with a standard tar command such as

```
tar xv
```

This command attempts to read from the default tape device on your system. If this command fails, you should contact your system administrator to find out the appropriate device name and then use a command such as:

```
tar xvf <device_name>
```

You should now have a compressed tar file of the form

```
ins++.ARCH.tar.Z
```

and a copy of the Insure++ FAQ in the current directory. ARCH is a shorthand label for your system, e.g. solaris or hp10. See the table on page 13 for a list of valid ARCH values. Please skip Step 2B and proceed directly to Step 3.

### Step 2B. Extract the CD-ROM contents

To mount the CD-ROM, you must be *root*.

```
su root
```

Create a `/cdrom` directory, if necessary

```
mkdir /cdrom
```

The actual mount command is different on each platform we support. Please use the appropriate command for your system. Note that the following commands assume your CD-ROM drive is at SCSI ID 6. If your drive is at a different SCSI location, substitute the appropriate device file name for your CD-ROM drive.

System	Command
IBM AIX	<code>mount -v cdrfs -r /dev/cd0 /cdrom</code>
DEC Alpha	<code>mount -t cdfs -o rrip /dev/rz6c /cdrom</code>

System	Command
HP-UX	<code>mount -F cdafs -r /dev/dsk/c0t6d0 /cdrom</code>
Linux	<code>mount -t iso9660 -o ro /dev/scd0 /cdrom</code>
SGI IRIX	<code>mount -t iso9660 -r /dev/scsi/sc0d610 /cdrom</code>
Solaris 2.x	<code>mount -F hsfs -r /dev/dsk/c0t6d0s2 /cdrom</code>

Change directory to the installation directory you created in Step 1.

```
cd /usr/local/parasoft
```

Then copy the tar file for your platform to the current directory with the following command. See the table in the section, "Installing Insure++" for a list of valid ARCH values

```
cp /cdrom/insure/tar/iARCH.tar .
```

Unmount the CD-ROM with the following command

```
umount /cdrom
```

You can now press the eject button on your CD-ROM drive to eject the CD before proceeding to Step 3.

### Step 3. Extract the installation script

Extract the installation script with the command

```
uncompress -c <tar_file> | tar xvf - install
```

for a compressed tar file or

```
gzip -dc <tar_file> | tar xvf - install
```

for a gzipped tar file in which the name of the compressed or gzipped tar file supplied to you should be inserted in place of the text <tar\_file>. If you are installing from a CD-ROM and your tar file is not compressed, use the following command:

```
tar xvf <tar_file> install
```

You are now ready to install Insure++ on your system using the provided installation script. Steps 4 through 11 will lead you through this procedure.

## Step 4. Install Insure++

Insure++ includes an installation script that will help you install Insure++ on your system. To run the installation script, execute the command

```
./install
```

The script first prints version and technical support information

```
Extracting installation scripts ...

Insure++ Installation Script Version 6.0 (3/18/01)
Copyright (C) 1997-01 by ParaSoft
Technical Support is available at:
E-mail:          support@parasoft.com
Web:             http://www.parasoft.com
Telephone:      (626) 305-0041
Fax:            (626) 305-9048
```

before asking you to confirm that you want to install Insure++ in the current directory. When the installation is completed, you will see the banner

```
Installation of Insure++ 6.0 completed
```

The Insure++ distribution consists of the following directories and files. ARCH below will be replaced in your distribution with your platform name, e.g. `sgi6`.

Directory	Contents
<code>bin.ARCH</code>	Insure++ executables
<code>lib.ARCH</code>	Insure++ libraries and interfaces
<code>src.ARCH</code>	Insure++ interface source code
<code>examples</code>	Insure++ example programs and scripts for customizing your use of Insure++
<code>insra</code>	Insra help files

Directory	Contents
Inuse	Inuse help files
tca	TCA help files
configure	Insure++ compiler configuration script
install	Insure++ installation script
FAQ.txt	Insure++ Frequently Asked Questions

## Step 5. Post-installation configuration

The installation script will next lead you through a series of questions as it configures Insure++ for your system.

- Which compilers will be used with Insure++.
- **Note:** This step does not tell Insure++ which compiler to use when instrumenting and compiling your source code. You will still need to add a `insure++.compiler <compiler>` option to one of your `.psrc` files to specify which compiler should be used by Insure++. Therefore, you should answer "yes" to these questions for any compiler which you might use with Insure++ at some point in the future.
- Whether to send output to `stderr` or `Insra` (a GUI report viewer) by default
- Whether to create a symbolic link to "insure" called "insight" for backwards compatibility (`insure` replaced `insight` in 4.1 as the driver program used to compile your files with Insure++)

If you want to configure Insure++ for additional compilers later, you can execute the script

```
./configure
```

in the installation directory.

Insure++ is now installed and configured on your system, but your system must be configured before use.

## Step 6. Install a license

After installing and configuring the necessary files, the installation script will look for a valid license for Insure++. If one is not found, it will ask if you would like to install one. If so, the script will start `pslic`, the ParaSoft License Manager.

If you get an error message from `pslic` saying that it cannot open a `.psrc` file, you should make sure that you have write permission in the `/usr/local/parasoft` (installation) directory and/or run `pslic` as superuser.

`pslic` will print out your machine and network id numbers. You should then phone, fax, or email this information to ParaSoft. You will receive a license which you can enter using `pslic`.

You can complete the remaining steps of the installation procedure without a license, but will not be able to use Insure++.

Once you have the license, run

```
pslic
```

Choose option "A" to add a license.

```
(A>dd a license
```

The first item you will need to enter is the network or host ID number, which should be the same number printed by `pslic`. Next, you will be prompted to enter the expiration date, which you received from ParaSoft. Finally, enter the password you were given. To complete the license installation, select option "E" to exit and save changes.

```
(E)xit and save changes
```

## Step 7. Set the PARASOFT environment variable (optional)

This step is optional. In most cases, you will not need to set this environment variable. However, you may find it useful as a shortcut to the Insure++ installation. Also, a tool may prompt you to set this environment variable.

To set the `PARASOFT` environment variable correctly, you will need to know the name of the directory in which Insure++ has been installed on

your system. Once you know this path you should define an environment variable called `PARASOFT` to be this pathname.

Typically, this can be performed by editing the file `.cshrc` in your home directory and adding a line similar to

```
setenv PARASOFT /usr/local/parasoft
```

## Step 8. Modify your PATH

You must add the directory containing the executables to your execution path. Normally, you do this by adding to the definition of either the `path` or `PATH` variables, according to the shell you are using.

The directory in which the executables are located will have a name that can be derived from the type of system you are running on, and is given to you by the install script.

A typical C-shell command would be

```
set path=($path /usr/local/insure/bin.linux)
```

if you are planning to use the Linux version of *Insure++*. Many systems have a built-in command called `arch`, which can be used to determine the name of the directory to put on your path as follows

```
set path=($path /usr/local/insure/bin.`arch`)
```

If you are in doubt as to which directory to put on your search path, ask your system administrator for help.

## Step 9. Redefine the LD\_LIBRARY\_PATH variable (Alpha, Linux, SGI, and Solaris only)

This step is optional, and is only necessary for users on the alpha4, linux, x86, sgi5, sgi6, and solaris platforms.

If you wish to use dynamic linking with *Insure++* on an Alpha running Digital UNIX 4.0, a PC running Linux, an SGI running IRIX 6.X, or a Sun running Solaris 2.X, you will need to modify your `LD_LIBRARY_PATH` environment variable to include the directory in which the *Insure++*

libraries are located. Other platforms can find the libraries automatically. You can set this properly by adding the appropriate path, for example,

```
/usr/local/parasoft/lib.sgi6
```

for sgi6, to the end of the definition of your `LD_LIBRARY_PATH` environment variable.

## Step 10. Modify your environment

After modifying the appropriate configuration files, you should execute the following commands to actually modify your working environment.

```
source ~/.cshrc  
rehash
```

## Step 11. Running an example

Change to the Insure++ examples directory and run the makefile.

```
cd $PARASOFT/examples/c  
make all
```



# Contacting ParaSoft

ParaSoft is committed to providing you with the best possible product support for Insure++. If you have any trouble installing or using Insure++, please follow the procedure below in contacting our Quality Consulting department.

- Check the manual.
- Be prepared to recreate your problem.
- Know your Insure++ version. (You can find it by typing `uname -a.`)
- If the problem is not urgent, then report it by email or by fax.
- If you call, please use a phone near your computer. The Quality Consultant may need you to try things while you are on the phone.

## Contact Information

- **USA Headquarters**  
Tel: (888) 305-0041  
Fax: (626) 305-9048  
Email: [quality@parasoft.com](mailto:quality@parasoft.com)  
Web Site: <http://www.parasoft.com>
- **ParaSoft France**  
Tel: +33 (0) 1 64 89 26 00  
Fax: +33 (0) 64 89 26 10  
Email: [quality@parasoft-fr.com](mailto:quality@parasoft-fr.com)
- **ParaSoft Germany**  
Tel: +49 (0) 78 05 95 69 60  
Fax: +49 (0) 78 05 95 69 19

Email: [quality@parasoft-de.com](mailto:quality@parasoft-de.com)

- **ParaSoft UK**

Tel: +44 020 8263 2827

Fax: +44 020 8263 2701

Email: [quality@parasoft-uk.com](mailto:quality@parasoft-uk.com)

# Running Insure++

The goal of this section is to give you enough information to start compiling and running your own programs under Insure++. Then you should be able to start finding bugs in your own software.

You use Insure++ by

1. Processing your program with the special `insure` program in place of your normal compiler. This creates a version of your code which includes calls to the Insure++ library and then passes it to your normal compiler.

If you simply re-link your code, you will get a basic level of checking for heap corruption and errors in calls to common C functions as well as checking for memory leaks. If you wish to do comprehensive checking, you can recompile your code with Insure++ for the strongest possible runtime checking.

2. Running the program in the normal manner.

**Note:** On Linux, Chaperon can be run on any executable file. It does not require any recompiling and relinking, or changing of environment variables. For more information see the section “Chaperon (Linux Only)” on page 53.

During the compilation process, Insure++ detects and reports various errors including:

- Illegal typecasts
- Incorrect parameters specified to library routines
- Memory corruption errors which are "obvious" at compile time

During execution, Insure++ reports on a wide variety of programming errors. For an exhaustive description of the types of errors detected, see the section “Insure++” on page 35. For each error reported, you will see the source line that appears to be incorrect and an explanation of what type of error occurred.

Normally, Insure++ sends its output to `stderr`, but there is also a graphical tool for viewing error messages, `Insra`. For more information, see “Insra” on page 84.

The easiest way to learn how to use Insure++ is to use it on an example program and see what it does. This section introduces two of the examples supplied with Insure++: one C example and one C++ example. You may want to copy the appropriate files to a directory and perform the steps as they are described.

The examples chosen here illustrate some of the simpler features of Insure++ and have been chosen to help you start using the system quickly. Once you have gone through this section, you should be in a position to use Insure++ on your own programs.

## Do it step-by-step

If you are working with a large application and don't want to jump right into recompiling all your code with Insure++, you can use the following method to integrate Insure++ into your development process step-by-step.

- Link your program with insure
- Add the option `insure++.summarize leaks` to your `.psrc` file and run your program. Leaked blocks with stack traces will be reported in a summary report at the conclusion of your program.
- To find out when and where the blocks were leaked as well as find more bugs, start recompiling parts of your code with Insure++
- Leak detection with linking alone does not work with `gcc/g++` under HP-UX or AIX

For an example of this procedure, see “Linking leak with Insure++” on page 25.

## A simple C example: sorting

The C demonstration code used in this section is a very simple program which attempts to sort an array of numbers.

If you wish to follow along with the example in this section, you can save some typing time by using the source code supplied with Insure++.

To get started, make a directory for temporary use and copy the source code for this example to it with commands similar to

```
mkdir $HOME/insure
cd $HOME/insure
cp /usr/local/insure/examples/c/bubble1.c .
```

## Compiling and running without Insure++

Once you've got a copy of the example program `bubble1.c` in your current directory, you can compile it with the command

```
cc -g -o bubble1 bubble1.c
```

and then run it from the shell in the normal manner.

```
bubble1
```

The program doesn't crash, and it doesn't print any error messages. Perhaps it's working?

Actually, this program has a serious error: a simple memory related bug.

## Compiling bubble1 with Insure++

Finding the error with Insure++ requires only that you recompile the program with the special `insure` command

```
insure -g -o bubble1 bubble1.c
```

`insure` simply replaces your normal compiler on the command line.

In minimal code instrumentation mode, the `insure` command is

```
insure -g -o bubble1 bubble1.c -Zmin
```

**Note:** The `-g` in both of the above commands is necessary on many platforms for Insure++ to be able to generate stack traces with file names and line numbers.

You can use the same options you normally use to compile and link your code by just replacing `cc` with `insure` in either your command lines or your `Makefile`.

You may also compile the source code with compilers such as `CC`, `gcc`, or `g++` with Insure++ instrumentation without modifying the `.psrc` file by doing the following: `insure <CC, gcc, or g++>`. For example, to compile `bubble1` in `CC` mode:

```
insure CC -g -o bubble1 bubble1.c
```

To compile `bubble1` in `gcc` mode:

```
insure gcc -g -o bubble1 bubble1.c
```

If you normally use a C compiler other than `cc`, you can make Insure++ do the same by creating a file called `.psrc` in your current directory and adding a line like `Insure++.compiler_c gcc`. This option tells Insure++ to use `gcc` instead of `cc` to compile C source files.

If your compiler is not directly supported by Insure++, you will also need to set the `compiler_acronym` option (see “Options used by Insure++” on page 179 for more details). If you need to use the `compiler_acronym` option, you will also need to use the `compiler` option instead of the `compiler_c` or `compiler_cpp` options. Both the `compiler` and `compiler_acronym` options override any `compiler_c` or `compiler_cpp` options.

If you are using a Makefile, things are often even easier, because many use the variable `CC` to define the name of the compiler to use. If this is true in your case, you can build a version of your program with Insure++ by typing the command

```
make CC=insure
```

You don't have to edit anything! Even better, you can build the original (unchecked) version or the Insure++ version by simply changing the command you type.

If your Makefile uses a different variable, e.g. `LD`, for the link command, you will need to use a command like

```
make CC=insure LD=insure.
```

## Running bubble1 with Insure++

Under minimal or normal code instrumentation mode, you run the program `bubble1.c` just as you would if you hadn't compiled it with Insure++.

```
bubble1
```

This time you get more interesting responses, shown below (normal code instrumentation mode).

```
[bubble1.c:20] **READ_BAD_INDEX**
>>     if(a[j-1] > a[j]) {

    Reading array out of range: a[j - 1]

    Index used : -1
    In block   : 0x0804cdc8 thru 0x0804cdef (40 bytes, 10 elements)
                vector, declared at bubble1.c, 10
    Stack trace where the error occurred:
                bubble_sort() bubble1.c, 27
                main() bubble1.c, 16
```

Here is what appears when minimal code instrumentation mode is used on bubble1.c.

```
[bubble1.c:20] **READ_BAD_INDEX**
>>     if(a[j-1] > a[j]) {

    Reading array out of range.

    Index used : -1 ** Using an element size of 4 (instead of 40)**
    In block   : 0x0804c684 thru 0x0804c6ab (40 bytes, 1 element)
    Stack trace where the error occurred:
                bubble_sort() bubble1.c, 27
                main() bubble1.c, 16
```

The output from Insure++ indicates that something is wrong and indicates the exact line number where the error occurs.

The error detected is indicated by the error code `READ_BAD_INDEX` and occurs at line 20 of `bubble1.c`. The line of code that causes the error is also shown along with a description of the problem. The other information shown in the display is best understood by examining the source code for the example, shown below.

```
1:      /*
2:      * File: bubble1.c
3:      */
4:      int vector[] = {4, 3, 6, 9, 1, 5, 8, 2, 0, 7};
5:
6:      main()
```

```

7:      {
8:          bubble_sort(vector,
9:                      sizeof(vector)/sizeof(vector[0]));
10:         exit(0);
11:     }
12:
13:     bubble_sort(a, n)
14:     int a[], n;
15:     {
16:         int i, j;
17:
18:         for(i=0; i<n; i++) {
19:             for(j=0; j<n-i; j++) {
20:                 if(a[j-1] > a[j]) {
21:                     int temp;
22:
23:                     temp = a[j-1];
24:                     a[j-1] = a[j];
25:                     a[j] = temp;
26:                 }
27:             }
28:         }
29:     }

```

We first declare an array which contains the list of values to be sorted in line 4. This is then passed from the main routine to the sorting subroutine in line 8.

The remaining information presented in the Insure++ bug report can now be interpreted as follows.

- The illegal index used in line 20 has the value  $-1$ , which implies that the variable  $j$  must have the value 0.
- The block of memory which is being accessed is fully described. Its starting and ending memory locations are given along with the size and number of elements in the array.
- The name of the array being accessed is given, including the location at which it was declared. Notice that this information describes the global variable `vector`, even though the `bubble_sort` routine is accessing this variable by the name `a`, as passed in its argument list.

- Finally, a stack trace is given which shows the sequence of function calls leading to this error.

From this information, it is hard to miss the cause of the problem in the code. The operation in line 20 is to compare an array element with its predecessor. This is the right operation to perform, but since we use the index values  $j$  and  $j-1$ , the loop range of  $j$  (in line 19) should start at 1, not 0.

This type of problem is very common and can easily go unnoticed in production code, because it doesn't crash the program and it may not even affect the result. In the example shown, the out-of-bounds value is quite likely zero, since it refers to another global variable.

## Eliminating the bug in bubble1

The fix in this case is particularly simple - line 19 should actually read

```
19: for(j=1; j<n-i; j++) {
```

To see that this actually fixes the problem, either modify the source file or copy `bubble2.c` from the Insure++ examples directory (see “A simple C example: sorting” on page 18). Compile and run it under Insure++ with the commands

```
insure -g -o bubble2 bubble2.c
bubble2
```

This time no errors are reported.

## Using Insure++ with C++ code

By default, Insure++ is set up to use the CC compiler with C++ source files on most platforms. The exceptions are Alpha (`cxx`) and RS/6000 (`x1C`). If you use a different compiler, you need to insert the line

```
insure++.compiler_cpp [cxx|CC|g++|x1C]
```

into your `.psrc` file. This tells Insure++ to compile all C++ source files with the given compiler.

Another important consideration when using Insure++ is source code file extensions. When Insure++ sees a `.c` file, it automatically treats it as C

code. Any file with a `.cc`, `.C`, `.ccp`, `.cxx` or `.c++` extension will be treated as C++ code. This is very important to understand. You cannot put C++ code in a file with a `.c` extension unless you also add the `insure++.c_as_cpp` on option to your `.psrc` file. For more information about this option, “Configuration Options” on page 175.

## Linking C++ objects with Insure++

If your makefile uses a separate link command with no source files on the link line, you must have the `insure++.compiler_default_cpp` option in your `.psrc` file to tell Insure++ to use C++ linkage. If Insure++ only sees objects and libraries on the link line, it cannot tell whether the code is C or C++. By default, it assumes it is C code and uses C linkage. The above option changes the default to C++ linkage.

As an alternative to the above method, if you use only C++ code, you can set the compiler option in your `.psrc` file to your C++ compiler, e.g. `compiler CC`. This option overrides any `compiler_c`, `compiler_cpp`, or `compiler_default` options present and tells Insure++ to use the indicated compiler every time it is called, for both compiling and linking.

## A C++ example: a memory leak

C++ can be a very difficult language in which to program, so we have significantly improved Insure++ to detect very hard-to-find bugs.

Often, code that contains serious errors can appear perfectly correct - at least until the problems start manifesting themselves in crashes, core dumps, or memory exhaustion. That is why we added capabilities like program tracing and detection of memory allocation conflicts, dead code, overloading operators and more.

To illustrate how Insure++ can detect tricky, well-disguised memory leaks in C++ code, let's consider the program whose source is presented below.

```

1:      /*
2:      * File: leak.C
3:      */
4:      #include <string.h>
5:

```

```

6:     union S1 {
7:         char *cp;
8:         S1() { cp = new char [10]; }
9:         S1(char *p) {
10:             cp = new char [10];
11:             strcpy(cp,p);
12:         }
13:         S1(S1 &s) {
14:             cp = new char [10];
15:             strcpy(cp,s.cp);
16:         }
17:         void mf(char *p) { strcpy(cp,p); }
18:     };
19:     void foo() {
20:         S1 s1,s2("Hello "),s3 = s2;
21:         s1.mf("SADF");
22:         s3.mf("World");
23:     }
24:     int main() {
25:         foo();
26:         return(0);
27:     }

```

## Linking leak with Insure++

Insure++ now detects memory leaks when the program is only linked with Insure++. Compiling and linking the example `leak.C` with the commands

```

CC -g -c leak.C
insure -g -o leak leak.o

```

and executing

```
leak
```

with `insure++.summarize` leaks in your `.psrc` file produces the output shown below in the leak summary report for the program `leak`.

```

***** INSURE++ SUMMARY ***** v5.0 ***
Program           : leak
Arguments         : Not available
Directory         : /users/trf/test
Compiled on      : Jan 05, 2001 16:13:59
Run on           : Jan 05, 2001 16:14:05
Elapsed time     : 00:00:00

```

```

*****
MEMORY LEAK SUMMARY
=====

3 outstanding memory references for 30 bytes.

Leaks detected at exit
-----
    10 bytes allocated
operator new()
           S1::S1() leak.C, 14
           foo() leak.C, 20
           main() leak.C, 25

    10 bytes allocated
operator new()
           S1::S1() leak.C, 10
           foo() leak.C, 20
           main() leak.C, 25

    10 bytes allocated
operator new()
           S1::S1() leak.C
           foo() leak.C, 20
           main() leak.C, 25

```

The output in tells us that there is a leak from each of the constructors in the `S1` class. In many cases this may be enough information to find and fix the bug. If it is not, however, Insure++ can give you more information, including where the leak actually occurred, not just where the leaked block was allocated.

## Compiling and running leak with Insure++

Compiling and linking the (`leak.C`) example with the command

```
insure -g -o leak leak.C
```

and executing

```
leak
```

produces the output shown below.

```
[leak.C:23] **LEAK_SCOPE**
>>     }

Memory leaked leaving scope: cp

Lost block : 0x00091ed0 thru 0x00091ed9 (10 bytes)
cp, allocated at:
operator new()
    S1::S1()    leak.C, 8
    foo()      leak.C, 20
    main()     leak.C, 25

Stack trace where the error occurred:
    foo()      leak.C, 23
    main()     leak.C, 25
```

The leak occurs because there is no destructor in `S1`. When `s1`, `s2`, and `s3` are called in `foo`, they appear to be on the stack, which would not cause memory to be allocated. However, `s1` calls `new` to allocate memory and does not have any way to deallocate it. This causes a large leak. Only the first leak is reported at runtime because by default Insure++ reports only one error per category per line. This behavior can be changed using the `insure++.report_limit .psrc` option.

## Eliminating the bug in leak

This error can be easily corrected by adding a destructor to `S1`. For example, adding the following line of code between lines 16 and 17 would eliminate the bug.

```
~S1() { delete[] cp; } //destructor
```

## Improving Insure++'s compile-time performance

If you are compiling in a remotely mounted directory, one easy way to decrease Insure++'s compile time is to use the `temp_directory` option. This `.psrc` option controls where Insure++ writes its temporary files during compilation. If you use it to redirect temporary files to a local disk,

compilation performance will improve dramatically. For example, adding the option

```
insure++.temp_directory /tmp
```

to your `.psrc` file tells Insure++ to write its temporary files in the `/tmp` directory.

You can significantly speed up the execution of your program by using the `header_ignore` option in your `.psrc` file to avoid instrumenting header files that you know are correct. See “Configuration Options” on page 175 for more information about this option.

## Performing a Quick Test With Chaperon (Linux only)

If you want to check your code for runtime errors but do not have the time to instrument your code with Insure++, you can check your code with Chaperon. Chaperon mode is faster-- though slightly less thorough-- than regular (Source Code Instrumentation) mode. You can run your application in Chaperon mode by entering

```
chaperon filename.exe
```

at the prompt.

For a complete description of Chaperon, including examples, see “Chaperon (Linux Only)” on page 53.

## Maintaining both normal and Insure++ builds

Another way to save time is to create a complete image of your project with Insure++ when you begin the debugging process. Then as you find and fix errors, you can just recompile one or two files at a time with Insure++. This will cut down greatly on compilation time in comparison with recompiling every file every time you want to switch from a normal build to an Insure++ build, or vice versa.

The `Makefile` shown below builds a program consisting of two source files, `func.c` and `main.c`. Typing `make` would build `main` in the current

directory, using the default settings in the `Makefile`. However, all that is necessary to build a completely separate version of the program with `Insure++` is the command

```
make CC=insure TDIR=insure
```

Alternatively, you can edit the `Makefile` to redefine `CC` and `TDIR` each time you want to switch between a normal and an `Insure++` build, if you prefer.

```
CC = cc
CFLAGS = -g
TDIR = .
OBJS = $(TDIR)/main.o $(TDIR)/func.o

$(TDIR)/main: $(OBJS)
    $(CC) $(CFLAGS) -o $(TDIR)/main $(OBJS)

$(TDIR)/main.o: main.c
    $(CC) $(CFLAGS) -o $(TDIR)/main.o -c main.c

$(TDIR)/func.o: func.c
    $(CC) $(CFLAGS) -o $(TDIR)/func.o -c func.c

clean:
    /bin/rm -rf $(TDIR)/*.o $(TDIR)/main
```

If you normally build libraries from your objects and do not add objects explicitly to your link line, you can do a similar trick by building a variable like `TDIR` into the object and library names, as shown in the `makefile` given in the figure below. In this case, a command like

```
make CC=insure TARGET=_ins
```

would leave you with versions of your objects, libraries, and executable tagged with names ending in `_ins`.

```
CC = cc
CFLAGS = -g
TARGET =
OBJS = main$(TARGET).o
LIBS = libfunc$(TARGET).a

main$(TARGET): $(OBJS) $(LIBS)
    $(CC) $(CFLAGS) -o main$(TARGET) $(OBJS) $(LIBS)

libfunc$(TARGET).a: func$(TARGET).o
```

```

ar ruv libfunc$(TARGET).a Func$(TARGET).o

main$(TARGET).o: main.c
    $(CC) $(CFLAGS) -o main$(TARGET).o -c main.c

func$(TARGET).o: func.c
    $(CC) $(CFLAGS) -o func$(TARGET).o -c func.c

clean:
    /bin/rm -rf *$(TARGET).o main$(TARGET)\
                                                libfunc$(TAR-
GET).a

```

## Common Insure++ options

Insure++ is an extremely customizable tool. While this flexibility is one of the great strengths of Insure++, it can present a problem for the new user. Although we ship Insure++ with defaults that will serve the majority of users quite well, we realize that some users have their own special needs and preferences. To help you configure Insure++ for your use, we would like to suggest some of the most popular options used by Insure++ users over the years and explain what they do. You can then pick and choose those that will be helpful in your particular situation.

More information about all of the options is available in the section “Configuration Options” on page 175. All of the options listed there can be placed in a file called `.psrc` in your local build directory with a prefix of `insure++`. They are applicable at different times in the build process.

## What bugs are uncovered by Insure++?

The programs `bubble2` and `leak` now run to completion, even when compiled with Insure++. Of course, `bubble1` and `leak` previously ran to completion, even though they contained the errors that Insure++ found. So what does it actually mean when Insure++ says there are no more errors?

Insure's testing is quite comprehensive. A program that passes Insure++ without any error messages will not contain any of the following:

- Uninitialized memory accesses
- Illegal pointer operations
- "Wild" pointer operations caused when a pointer skips from one data object to point at another
- Dynamic memory errors
- Accesses of memory blocks outside their legal bounds
- Memory leaks

Note that this is only a brief summary. The full set of errors detected by Insure++ is described in the section "Insure++" on page 35 and listed in the section "Error Codes" on page 199.

# Preventing Errors With CodeWizard

You can run Insure++ with Code Wizard to perform both automatic error detection (Insure++) and automatic error prevention (CodeWizard) in a single step. CodeWizard (available separately from ParaSoft) checks your code for design and coding problems that can lead to bugs and other problems later on. By finding and fixing problems early, you can save yourself untold amounts of debugging and maintenance time. In addition, you will be learning valuable coding techniques that can actually reduce the number of bugs in your code in the future.

By combining the analytical power of Code Wizard with Insure++'s bug-finding prowess you can speed up your entire development process. At once, you'll see where the bugs are in your program and where trouble is likely to occur in the future. Fix all the errors now and you'll save yourself time and headaches later.

You can download CodeWizard now at:

<http://www.parasoft.com/products/devtools.htm>.

# Analyzing Code Coverage With TCA

The Total Coverage Analysis (TCA) add-on works hand-in-hand with Insure++ and reports which parts of your program have actually been tested by Insure++ and how often each block of code was executed. Using TCA with Insure++ can dramatically improve the efficiency of your testing and guarantee faster delivery of more reliable programs.

## To run TCA

Insure++ tracks code coverage information in a file called `tca.log`. This file is located in the same directory as your program.

You can analyze Insure++ coverage from the command line or with the TCA add-on GUI.

To review code coverage information from the TCA GUI

1. Click **File> Load** in the TCA window and select the `tca.log` file located in your program's directory.

To review code coverage information, type:

```
tca tca.log
```

For command line options, type:

```
tca
```

**Note for Linux:** TCA does not work with Chaperon. Since Chaperon works on the original executable (non-instrumented) and `tca.map` is not even created, there will be no `tca.log`.

# Optimizing Dynamic Memory With Inuse

Inuse is a graphical Insure++ add-on that allows you to watch how your programs handle memory 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:

- Find memory leaks.
- See how much memory an application uses in response to particular user events.
- Look for memory fragmentation to see if different allocation strategies might improve performance.

## To run Inuse

In normal use, you should enter the `inuse` command once and simply leave it running as a background process.

```
inuse
```

Inuse will be run the next time Insure++ is run. For more information, see “Running Inuse” on page 140.

# Insure++

As shown in the "Getting Started" section, 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; this report usually contains enough detail to track down and correct the problem.

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 shipped to customers and they run it on *their* test cases. 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. The following sections describe 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 which concatenates the arguments given on the command line and prints the resulting string

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



```
>> 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, 7

Stack trace where the error occurred:
    main() hello.c, 16
```

```
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 (i.e., a global variable), locally on the stack, dynamically (with `malloc` or `new`), or even as a shared memory block.

It also detects the case in which 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.

```
1      /*
2      * File: hello2.c
3      */
```

```

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

```

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

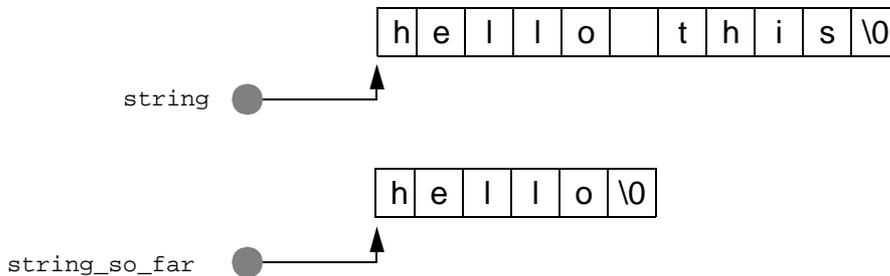
A “memory leak” occurs when a piece of dynamically allocated memory cannot be freed because the program no longer contains any pointers that point to the block. A simple example of this behavior can be seen by running the (corrected) “Hello world” program with the arguments

```
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.

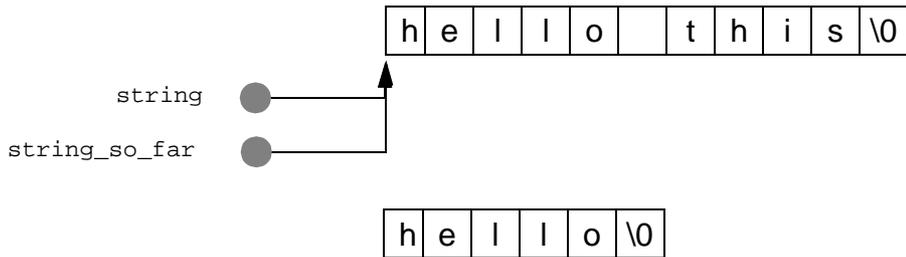


### Pointer assignments before the memory leak

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:27] **LEAK_ASSIGN**
>>                 string_so_far = string;
```

Memory leaked due to reassignment: string

In block: 0x0001fbb0 thru 0x0001fbb6 (7 bytes)  
 block allocated at:  
     malloc() (interface)  
     main() hello3.c, 17

Stack trace where the error occurred:  
     main() hello3.c, 27

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:

<code>LEAK_FREE</code>	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++.
<code>LEAK_RETURN</code>	Occurs when a function returns a pointer to an allocated block of memory, but the returned value is ignored in the calling routine.
<code>LEAK_SCOPE</code>	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.

## Should memory leaks be fixed?

Whether or not this is a serious problem depends on your application. To get more information on the seriousness of the problem, check the **Summarize Leaks** box in the **Reports** tab of the Insure++ Control Panel.

When you run the program again, you will see the same output as before, followed by a summary of all the memory leaks in your code.

```
MEMORY LEAK SUMMARY
=====

4 outstanding memory references for 55 bytes.
```

```
Leaks detected during execution
-----
55 bytes 4 chunks allocated at hello3.c, 17
```

This shows that even this short program lost four different chunks of memory. The total of 55 bytes isn't very large and you might ignore it in a program this size. If this was a routine in a larger program, it would be a serious problem because every time the routine is called it allocates blocks of memory and loses some. As a result, the program gradually consumes more and more memory and will finally crash when the memory space on the host machine is exhausted.

This type of bug can be extremely hard to detect, because it might take literally days to show up.

**Note:** You may be wondering why Insure++ only prints one error message although the summary indicates that four memory leaks occurred. This is because Insure++ normally shows only the first error of any given type at each particular source line. If you wish, you can change this behavior as described in "Displaying repeated errors" on page 70.

## Additional Information

You can obtain additional information about memory leaks and outstanding memory by enabling the summaries for leaks and outstanding memory in the **Reports** tab of the Insure++ Control Panel. A dynamically allocated memory block is categorized as a leak if a pointer to that block is lost during program execution. A block is categorized as outstanding memory if a pointer to the block is retained up to program termination, but the block is not freed prior to program termination.

## Finding all memory leaks

For an even higher level of checking, we suggest the following algorithm for removing all memory leaks from your code.

1. Run your program from Inuse (See "Running Insure++" on page 17). If you see an increase in the heap size as you run the program, you are leaking memory.

2. Compile all source code, but not libraries, with Insure++. Clean all leaks that are detected by Insure++.
3. Compile everything that makes up your application with Insure++ -- source code and libraries. Clean any leaks detected by Insure++. If you do not have source for any of the libraries, skip this step and proceed to Step 4.
4. Repeat Step 1. If memory is increasing, check **Summarize Outstanding Memory** in the **Reports** tab of the Insure++ Control Panel and run your Insure++-checked program again. Any outstanding memory reference shown is a potential leak.
5. You must now examine each outstanding memory reference to determine whether or not it is a leak. If the pointer is passed into a library function, it may be saved. If this is the case, it is not a leak. Once every outstanding memory reference is understood, and those that are leaks are cleared, the program is free of memory leaks.

## 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:

```

21:         if(string_so_far != (char *)0) {
22:             free(string_so_far);
23:             strcpy(string, string_so_far);
24:         }
25:         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

- 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 (See “`RETURN_FAILURE`” on page 324). 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, since 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 35. Another common problem is caused by trying to work with strings that are not null-terminated, as in the following example.

```

1:      /*
2:      * File: readovr2.c
3:      */
4:      int main()
5:      {
6:          char junk;
7:          char b[8];
8:          strncpy(b, "This is a test",
9:                sizeof(b));
10:         printf("%s\n", b);
11:         return (0);
12:     }

```

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.

- `copy` - Normally, Insure++ doesn't complain when you assign a variable using an uninitialized value, since 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.
- `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: readunil.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, since 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, since 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. See “`UNUSED_VAR`” on page 328 for more details.

## Data representation problems

A lot of 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

```
char *p;  
int ip;  
  
ip = (int)p;
```

On many systems this type of operation would be valid and would not cause any problems. However, when such code is ported to alternative architectures problems can arise. 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 DEC Alpha (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, e.g.,

```
int myblock[128];
```

but as a pointer in another

```
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`

```
foo()
{
    float f;

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

corrupts memory, since too much data will be written over the supplied variable. This error can be very difficult to detect.

A more subtle issue arises when data types used in I/O statements “accidentally” match. The code

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

functions correctly on machines where types `int` and `long` have the same number of bits, but fails otherwise. 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 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.

**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.

Insure++ detects the error in the following program

```

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

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

```

in which the argument passed to the function `foo` in `main` is an integer rather than a floating point number.

**Note:** Converting this program to ANSI style (e.g., 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.

## C++ compile time warnings

During compilation, Insure++'s parser detects a number of C++-specific problems and prints warning messages. These messages are coded by the chapter, section, and paragraphs pertaining to that warning in the draft ANSI standard. Therefore, if you are uncertain what a particular warning message means or would like additional information, you can consult the standard for an explanation.

As an example, when processed by Insure++, the code

```
void foo(char *str) { }
void func()
{
    void *iptr = (char *) 0;
    foo(iptr);
}
```

will produce the warning

```
insure -c foo.cpp
[foo.cpp:5] Warning:13-2: wrong arguments passed to function 'foo'
|   declared at: [foo.cpp:1]
|   expected args: (char *)
|   passed args: (void *)
>> foo(iptr);
```

## 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 code:

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

would generate an error since the last argument passed to the `fseek` function is outside the legal range.

## 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. It is a very rare program that deals 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. Normally these errors are suppressed, since it is assumed that the application is handling them itself, but they can be enabled at runtime by unsuppressing

```
RETURN_FAILURE
```

in the Suppressions Control Panel (accessed by clicking the **Suppressions** button in the **Reports** tab of the Insure++ Control Panel). 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 are automatic detection of errors in the following situations

- `malloc` runs out of memory.
- Files that don't 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.

# Chaperon (Linux Only)

## Introduction

Chaperon checks all data memory references made by a process, whether in the developer's compiled code, language support routines, shared or archive libraries, or operating system kernel calls. Chaperon 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 works with existing executable programs. In most cases, Chaperon requires no recompilation and no relinking, and no changes to environment variables. Just add `Chaperon` to the beginning of the command line; Chaperon will run the process and check all data memory references.

When Chaperon detects improper behavior, it issues an error message identifying the kind of error and where it occurred. Improper behavior is any access to a logically unallocated region, a Read (or Modify) access to bytes which have been allocated but not yet Written, or attempts to free the same block twice.

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 unreachable 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 oracle to specify its address as the parameter to `free()`. At `exit()` Chaperon reports memory leaks automatically, and reports blocks in use if requested by the command line option:

```
Chaperon %program_name%
```

Using Chaperon does not require running under a debugger, but Chaperon also works with existing debuggers such as `gdb`. For more information, see "Using Chaperon With `gdb`" on page 64.

## Requirements and limitations

### Requirements

- ELF format executables and shared libraries, with `/lib/ld-linux.so.2` -> `ld-2.1.1.so` (or compatible), as the ELF program interpreter. The important interfaces are `_dl_runtime_resolve`, `_dl_relocate_object`, `_dl_debug_state`, and `_r_debug`.
- Any x86 processor [ $x \geq 3$ ] running Linux. In case of opcode conflict between manufacturers, Chaperon follows the Intel documentation.
- Linux kernel 2.2.5 or compatible. Other kernels will work with adjustment of the accounting for system calls.
- No kernel threads. There must be only one actual execution context per process, and Chaperon must see all the switching between register sets.
- `malloc/free/etc` must not call `sigaction` that gets used.
- `vfork()` is remapped to `fork()`. Programs depending on `vfork()` semantics may not work properly.
- 32-bit code (no `0x67` address size prefix; but `0x66` operand size prefix is OK), flat model. Any explicit `cs`, `ds`, or `ss` segment selector in the instruction stream must equal the corresponding current actual selector. Chaperon's access accounting treats all offsets as belonging to segment `ds`. Application code using `es`, `fs`, or `gs` does run; but the accounting may become confused.

### Bitfields

Chaperon accounts for memory on a byte-by-byte basis. Since the mapping between bytes and bitfields need not be 1-to-1 and onto, there are problems. By default, Chaperon uses heuristics to guess that some instruction sequences (that would otherwise generate complaints of Read before Write) correspond to legitimate bitfield operations, and the heuristics enable Chaperon to suppress those complaints. They also cause a whole byte to be marked as Written as soon as the first write to

any bitfield that intersects it. The heuristics are not complete; there will still be "false positive" complaints of Read before Write. Some source statements and expressions that are not bitfields can generate code that looks like bitfields, and for which the heuristics should be disabled; use the command line parameter `-bitfields=0`. The general palliative for cleaning up Chaperon complaints about bitfields is to clear all words that contain bitfields as soon as the memory is allocated, perhaps using `memset` and perhaps employing a `union`. This can even be more efficient, but some programmers consider it to be distasteful.

## Symbols, tracebacks, and compilers

Chaperon's underlying execution engine and tracking for memory state depend only on x86 architecture, and are compiler independent. But the generation and reporting of tracebacks and symbols relies on Chaperon being able to find and identify the code and symbols. Chaperon recognizes the functions involved in dynamic binding (`.dynsym` symbols), static binding (`.symtab` symbols), and one flavor of debugging (`.stab` symbols). Chaperon also handles the anonymous functions it finds by taking the transitive closure of the static lexical call graph, and their immediate successors in the `.text` section. (In most cases, Chaperon correctly separates and identifies anonymous functions that `gdb` lumps together with the preceding named function.) For modules that have been stripped of symbols, this can omit functions that are called only indirectly through a pointer or table, perhaps including some C++ virtual functions. The remedy is not to strip such modules of their symbols.

The `jmp` table code generated for `switch` statements also can be problematic for Chaperon to understand, and may require updates to accommodate different compilers.

## System calls

Chaperon checks the documented memory access behavior of kernel calls for Linux 2.2.x except `bdflush`, `capget`, `capset`, `getpmsg`, `ipc` (but `shm*` shared memory calls *are checked*), `modify_ldt`, `nfsservctl`, `prctl`, `putpmsg`, `quotactl`, `sysfs`, and `vm86*`. Chaperon understands the "regular" `SYS_ioctl` calls whose command word uses `_IOR`, `_IOW`, or

`_IOWR`, plus important non-regular cases such as `TIOC*` (terminal control) and `SIOC*` (socket control).

## Space

Chaperon runs in the same execution context and address space as the process that Chaperon is checking. The linear coefficients of space overhead are 2 bits of accounting info per byte of address space used by the application, plus  $(16 + 8 * \text{traceback\_length})$  bytes per active allocated block. Process sizes greater than about 500MB have not been well explored.

## Memory states and access accounting

State	Read or Modify Access	Write Access
Unallocated	error: Read before Allocate	error: Write before Allocate
Allocated but not Written	error: Read before Write	OK; becomes Allocated and Written
Allocated and Written	OK	OK

See also “Bitfields” on page 54.

**Allocators:** `malloc`, `calloc`, `realloc`, `memalign`, `__libc_malloc`, `__libc_calloc`, `__libc_realloc`, `__libc_memalign`, **stack growth** (push, create frame), `__brk`, `brk`, `__sbrk`, `sbrk`, `mmap`

**De-allocators:** `free`, `realloc`, `__libc_free`, `__libc_realloc`, `__brk`, `brk`, `__sbrk`, `sbrk`, **stack trim** (pop, delete frame), `munmap`

**Other known functions:** `memcpy`, `memset`, `memmove`, `memchr`, `bcopy`, `bzero`, `strcat`, `strchr`, `__stpcpy`, `strcpy`, `strrchr`. These are optimized for faster performance, and/or to reduce the clutter of multiple error messages that arise from a single call, and/or to suppress “false positive”

Read before Write messages from instruction sequences that are known to be used to implement write-allocate cache control, or speculative word-wide reading of byte arrays.

Handling of `realloc(ptr, size)`:

```
If 0==size then free(ptr);
else if 0==ptr then mloc(size);
else {free(ptr); malloc(size)}
```

and arrange for the new contents of the `malloc()`ed region to equal the old contents for the first `min(old_size, new_size)` bytes.

## Examples

**Note:** The numeric values of addresses might not match when the examples are re-run. For instance, locations in the stack (`0xbfffffff` and lesser nearby locations) depend on the number of characters in environment variables. Locations in shared libraries (`0x40000000` and greater nearby locations) change with different versions and different values of `LD_PRELOAD`. Locations in application code (`0x08040000` and greater nearby locations) depend on compiler and compiler options.

## WRITE\_OVERFLOW

To run this example, first, compile the code with `gcc`.

```
gcc -g -o writover writover.c
```

Run the new executable on Chaperon.

```
Chaperon writover
```

Chaperon should report errors such as:

```
#----- writover.c
#include <stdlib.h>

int
main()
{
    /* An example of Write before Allocate */
    char *p = malloc(10);
    p[11] = 3;
    return 0;
}
```

```

#-----
$ $HOME/bin/Chaperon ./writover
# Chaperon banner and license info
// Chaperon(tm) memory access checker version 2.0 2000-07-07.
// Copyright 1999 BitWagon Software LLC. All rights reserved.
// Copyright 2000 ParaSoft Corp. All rights reserved.
[writover.c:8] **WRITE_OVERFLOW**
>> p[11] = 3;

Writing overflows memory.

          bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
          |                10                | 1 | 1 |
                                               wwwwww

Writing (w) : 0x0804958b thru 0x0804958b (1 byte)
To block (b) : 0x08049580 thru 0x08049589 (10 bytes)
      block allocated at writover.c, 7
                main() writover.c, 7
      __libc_start_main() ../sysdeps/generic/libc-
start.c, 92

Stack trace where the error occurred:
                main() writover.c, 8
      __libc_start_main() ../sysdeps/generic/libc-
start.c, 92

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

Exit with return code 0 (0x0000).
  15 total blocks allocated
   0 total blocks freed.
Chaperon searching memory blocks...

End of memory leak processing.

```

## READ\_UNINIT\_MEM

```

#----- readunin.c
int
main()
{
    int x;
    if (3==x) {
        return 5;
    }
}

```

```

    }
    else {
        return 7;
    }
}
#-----
// Chaperon(tm) memory access checker version 2.0 2000-07-07.
// Copyright 1999 BitWagon Software LLC. All rights reserved.
// Copyright 2000 ParaSoft Corp. All rights reserved.
[readunin.c:5] **READ_UNINIT_MEM(read)**
>>         if (3==x) {

Reading uninitialized memory.

Address: 0xbffff04
In block: 0xbffff000 thru 0xbfffffff (8192 bytes)
           stack

Stack trace where the error occurred:
           main() readunin.c, 5
           __libc_start_main() ../sysdeps/generic/libc-
start.c, 92

Exit with return code 7 (0x0007).
  14 total blocks allocated
   0 total blocks freed.
Chaperon searching memory blocks...

End of memory leak processing.
Exit 7

```

## FREE\_DANGLING

```

#----- freedngl.c

#include <stdlib.h>

int
main()
{
    char *p = malloc(13);
    free(p);
    free(p);
    return 0;
}

```

```

}
#-----
// Chaperon(tm) memory access checker version 2.0 2000-07-07.
// Copyright 1999 BitWagon Software LLC. All rights reserved.
// Copyright 2000 ParaSoft Corp. All rights reserved.
[freedngl.c:8] **FREE_DANGLING**
>> free(p);

Freeing dangling pointer.

Pointer      : 0x080495c8
In block     : 0x080495c8 thru 0x080495d4 (13 bytes)
               block allocated at freedngl.c, 6
               main() freedngl.c, 6
               __libc_start_main() ../sysdeps/generic/libc-
start.c, 92

stack trace where memory was freed:
               main() freedngl.c, 7
               __libc_start_main() ../sysdeps/generic/libc-
start.c, 92

Stack trace where the error occurred:
               main() freedngl.c, 8
               __libc_start_main() ../sysdeps/generic/libc-
start.c, 92

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

Exit with return code 0 (0x0000).
 15 total blocks allocated
  1 total blocks freed.
Chaperon searching memory blocks...

End of memory leak processing.

```

## Summarize leaks

Make sure that your .psrc file has the following line in it:

```
insure++.summarize leaks outstanding
```

Then run Chaperon again.

```
$ $HOME/bin/Chaperon ./writover
```



```

        dl_main()      rtld.c, 632
    _dl_sysdep_start() ../sysdeps/generic/dl-sysdep.c, 171
    _dl_start_final()  rtld.c, 232
        _dl_start()    rtld.c, 193
    0x40101996()

556 bytes      1 chunk allocated at dl-minimal.c, 87
        calloc()      dl-minimal.c, 87
        _dl_new_object() dl-object.c, 40
    _dl_map_object_from_fd() dl-load.c, 795
        _dl_map_object() dl-load.c, 1416
    _dl_map_object_deps() dl-deps.c, 218
        dl_main()      rtld.c, 632
    _dl_sysdep_start() ../sysdeps/generic/dl-sysdep.c, 171
    _dl_start_final()  rtld.c, 232
        _dl_start()    rtld.c, 193
    0x40101996()

556 bytes      1 chunk allocated at dl-minimal.c, 87
        calloc()      dl-minimal.c, 87
        _dl_new_object() dl-object.c, 40
        dl_main()      rtld.c, 508
    _dl_sysdep_start() ../sysdeps/generic/dl-sysdep.c, 171
    _dl_start_final()  rtld.c, 232
        _dl_start()    rtld.c, 193
    0x40101996()

288 bytes      3 chunks allocated at dl-minimal.c, 87
        calloc()      dl-minimal.c, 87
    _dl_check_map_versions() dl-version.c, 273
    _dl_check_all_versions() dl-version.c, 365
    version_check_doit()    rtld.c, 299
    _dl_receive_error()    dl-error.c, 169
        dl_main()      rtld.c, 632
    _dl_sysdep_start() ../sysdeps/generic/dl-sysdep.c, 171
    _dl_start_final()  rtld.c, 232
        _dl_start()    rtld.c, 193
    0x40101996()

22 bytes      1 chunk allocated at dl-load.c, 312
    add_name_to_object()    dl-load.c, 312
        _dl_map_object() dl-load.c, 1291
    _dl_map_object_deps() dl-deps.c, 218
        dl_main()      rtld.c, 632
    _dl_sysdep_start() ../sysdeps/generic/dl-sysdep.c, 171
    _dl_start_final()  rtld.c, 232
        _dl_start()    rtld.c, 193
    0x40101996()

18 bytes      1 chunk allocated at dl-object.c, 41
        _dl_new_object() dl-object.c, 41
    _dl_map_object_from_fd() dl-load.c, 795
        _dl_map_object() dl-load.c, 1416
    _dl_map_object_deps() dl-deps.c, 218
        dl_main()      rtld.c, 632
    _dl_sysdep_start() ../sysdeps/generic/dl-sysdep.c, 171
    _dl_start_final()  rtld.c, 232
        _dl_start()    rtld.c, 193

```

```

0x40101996()

15 bytes      1 chunk allocated at dl-load.c, 140
    _dl_map_object() dl-load.c, 140
    _dl_map_object_deps() dl-deps.c, 218
        dl_main() rtld.c, 632
    _dl_sysdep_start() ../sysdeps/generic/dl-sysdep.c, 171
    _dl_start_final() rtld.c, 232
    _dl_start() rtld.c, 193
    0x40101996()

15 bytes      1 chunk allocated at dl-object.c, 92
    _dl_new_object() dl-object.c, 92
    _dl_map_object_from_fd() dl-load.c, 795
        _dl_map_object() dl-load.c, 1416
    _dl_map_object_deps() dl-deps.c, 218
        dl_main() rtld.c, 632
    _dl_sysdep_start() ../sysdeps/generic/dl-sysdep.c, 171
    _dl_start_final() rtld.c, 232
    _dl_start() rtld.c, 193
    0x40101996()

12 bytes      1 chunk allocated at dl-deps.c, 437
    _dl_map_object_deps() dl-deps.c, 437
        dl_main() rtld.c, 632
    _dl_sysdep_start() ../sysdeps/generic/dl-sysdep.c, 171
    _dl_start_final() rtld.c, 232
    _dl_start() rtld.c, 193
    0x40101996()

12 bytes      1 chunk allocated at dl-load.c, 528
    _dl_init_paths() dl-load.c, 528
        dl_main() rtld.c, 632
    _dl_sysdep_start() ../sysdeps/generic/dl-sysdep.c, 171
    _dl_start_final() rtld.c, 232
    _dl_start() rtld.c, 193
    0x40101996()

9 bytes       1 chunk allocated at dl-object.c, 41
    _dl_new_object() dl-object.c, 41
        dl_main() rtld.c, 508
    _dl_sysdep_start() ../sysdeps/generic/dl-sysdep.c, 171
    _dl_start_final() rtld.c, 232
    _dl_start() rtld.c, 193
    0x40101996()

8 bytes       1 chunk allocated at dl-sysdep.c, 292
    _dl_important_hwcaps() ../sysdeps/generic/dl-sysdep.c, 292
    _dl_init_paths() dl-load.c, 524
        dl_main() rtld.c, 632
    _dl_sysdep_start() ../sysdeps/generic/dl-sysdep.c, 171
    _dl_start_final() rtld.c, 232
    _dl_start() rtld.c, 193
    0x40101996()

```

## Using Chaperon With gdb

Chaperon can be used with your existing gdb, or with a modified gdb-5.0, provided by ParaSoft in `INSTALLDIR/bin.linux2/gdb.exe`.

Either version can be used to set a breakpoint in `_Insure_trap_error`, which allows you to stop program execution at a point where a memory reference error is detected, and examine program state, values of variables, etc.

The ParaSoft version of gdb could also be used to set breakpoints in your binary and execute gdb commands, such as `next`, `step`, `continue`.

**Note:** Your existing gdb will be able to set breakpoints in your executable as well, but you will not be able to properly continue execution after the breakpoint.

For example:

```
$ gdb.exe Chaperon
GNU gdb 5.0 extended 2000-09-12 by ParaSoft Corporation for Chaperon on
Linux x86
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License ...
(gdb) b gdb_setup
Breakpoint 1 at 0x1700
(gdb) run ./chaptest
Starting program: Chaperon ./chaptest
// Chaperon(tm) memory access checker version 2.0 2000-09-25.
// Copyright 1999 BitWagon Software LLC. All rights reserved.
// Copyright 2000 ParaSoft Corp. All rights reserved.

Breakpoint 1, 0x1700 in gdb_setup () from ./chaptest
(gdb) b main
Breakpoint 2 at 0x80483d6: file chaptest.c, line 4.
(gdb) disable 1
gdb) cont

Continuing.

Breakpoint 2, main () at chaptest.c:4
4      char *p = (char *)malloc(20);
(gdb) n
6      if (p[1]) {
(gdb) b _Insure_trap_error
Breakpoint 3 at 0x1706
(gdb) c
Continuing.
[chaptest.c:6] (Thread 0) **READ_UNINIT_MEM(read)**
>>     if (p[1]) {
```

```
Reading uninitialized memory.
```

```
Address: 0x080495b1
In block: 0x080495b0 thru 0x080495c3 (20 bytes)
           block allocated at chaptest.c, 4
           main() chaptest.c, 4
```

```
Stack trace where the error occurred:
main() chaptest.c, 6
```

```
Breakpoint 3, 0x1706 in _Insure_trap_error () from ./chaptest
```

```
(gdb) where
#0 0x1706 in _Insure_trap_error () from ./chaptest
#1 0x80483e9 in main () at chaptest.c:6
#2 0x4012d9cb in __libc_start_main (main=0x80483d0 <main>, argc=1,
  argv=0xbffff7b4, init=0x8048298 <_init>,
  fini=0x804845c <_fini>, rtdl_fini=0x4010ae60 <_dl_fini>,
  stack_end=0xbffff7ac) at ../sysdeps/generic/libc-start.c:92
(gdb) quit
```

Finally, you can download the `gdb-5.0.patch` file, which contains diffs against the original `gdb-5.0` source, at at:

```
http://www.parasoft.com/products/insure/manuals/v5\_2/
  unix/ users/chapsamp.html
```



# Reports

The error reports that have already been shown indicate that Insure++ provides a great deal of information about the problems encountered in your programs. Insure++ also provides many ways of customizing the presentation of this information to suit your needs.

## Default behavior

By default, Insure++ adopts the following error reporting strategy:

- Error messages are “coded” by a single word shown in upper-case, such as `HEAP_CORRUPT`, `READ_OVERFLOW`, `LEAK_SCOPE`, etc.
- Messages about error conditions are displayed unless they have been suppressed by default or in a site specific configuration file. (See “Error Codes” on page 199 for a list.)
- Only the first occurrence of a particular (unsuppressed) error at any given source line is shown. (See “Report summaries” on page 77 for ways to change this behavior.)
- Error messages are sent to the console (`stderr`), the Insra GUI, or to a separate report file. (See “The report file” on page 67 or “Sending messages to Insra” on page 87).
- Each error shows a stack trace of the previous routines, displayed all the way back to your `main` program.

## The report file

Normally, error reports are displayed on the UNIX `stderr` I/O stream. Users interested in sending output to Insra should consult the section “Insra” on page 84. If you wish to send both your program’s output and the Insure++ reports to a file, you can use the normal console redirection method. An alternative is to have Insure++ redirect only its output directly by adding an option similar to

```
insure++.report_file bugs.dat
```

to your `.psrc` file. This tells Insure++ to write its reports to the file `bugs.dat`, while allowing your program's output to display as it normally would. Whenever this option is in effect you will see a "report banner" similar to

```
** Insure++ messages will be written to bugs.dat **
```

on your terminal when your program starts to remind you that error messages are being redirected. To suppress the display of this banner add the option

```
insure++.report_banner off
```

to your `.psrc` file.

Normally the report file is overwritten each time your program executes, but you can force messages to be appended to an existing file with the command

```
insure++.report_overwrite false
```

If you want to keep track of the reports from multiple runs of your code, an alternative is to have Insure++ automatically generate filenames for you based on a template that you provide. This takes the form of a string of characters with tokens such as `%d`, `%p`, or `%V` embedded in the template. Each of these is expanded to indicate a certain property of your program as indicated in the table in the section "Configuration Options" on page 175.

For example, the option:

```
report_file %v-errs.%D
```

when executed with a program called `foo` at 10:30 a.m. on the 21st of December 2001, might generate a report file with the name

```
foo-errs.20011221103032
```

(The last two digits are the seconds after 10:30 on which execution began.)

**Note:** Programs which fork will automatically have a `-%n` added to their format strings unless a `%n` or `%p` token is explicitly added to the format string by the user. This ensures that output from different processes will always end up in different report files.

You can also include environment variables in these filenames so that

```
$HOME/reports/%v-errs.%D
```

generates the same filename as the previous example, but also ensures that the output is placed in the reports sub-directory of the user's HOME.

This method is very useful for keeping track of program runs during development to see how things are progressing as time goes on.

## Customizing the output format

By default, Insure++ displays a particular banner for each error report, which contains the filename and line number containing the error, and the error category found, e.g.,

```
[foo.c:10] **READ_UNINIT_MEM(copy)**
```

If you wish, you can modify this format to suit either your aesthetic tastes or for some other purpose, such as enabling the editor in your integrated environment to search for the correct file and line number for each error.

Customization of this output is achieved by setting the `error_format` option in your `.psrc` file to a string of characters containing embedded tokens which represent the various pieces of information that you might wish to view (“Options used by Insure++” on page 179).

For example, the command

```
error_format "\"%f\", line %l: %c"
```

would generate errors in the following format:

```
"foo.c", line 8: READ_UNINIT_MEM(copy)
```

which is a form recognized by editors such as GNU Emacs.

**Note:** Notice how the embedded double quote characters required backslashes to prevent them being interpreted as the end of the format string.

A multi-line format can also be generated with a command such as

```
error_format "%f, line %l\n\t%c"
```

which might generate

```
foo.c, line 8
      READ_UNINIT_MEM(copy)
```

## Displaying process information

When using Insure++ with programs which fork into multiple processes, you might wish to display additional process-related information in your error reports. For example, adding the option

```
insure++.error_format <all>
"%f, line %l: \n\tprocess %p@%h: %c"
```

in your `.psrc` file would generate errors in the form

```
foo.c, line 8:
    process 1184@gobi: READ_UNINIT_MEM(copy)
```

which contains the name of the machine on which the process is running and its process ID.

## Displaying the time at which the error occurred

It is often convenient to know exactly when various errors occurred. You can extend the error reports generated by Insure++ in this fashion by adding the `%d` and/or `%t` characters to the error report format as specified in your `.psrc` file. For example, the format

```
insure++.error_format "%f:%l, %d %t <%c>"
```

generates error reports in the form

```
foo.c:8, 9-Jan-2001 14:24:03 <READ_NULL>
```

## Displaying repeated errors

The default configuration suppresses all but the first error of any given kind at a source line. You can display more errors by modifying the `.psrc` file in either your working or `HOME` directory.

For example, adding the line

```
insure++.report_limit 5
```

to your `.psrc` file will show the first five errors of each type at each source line.

Setting the value to zero suppresses any messages except those shown in summaries (see “Report summaries” on page 77).

Setting the `report_limit` value to -1 shows all errors as they occur.

Note that not all information is lost by showing only the first (or first few) errors at any source line. If you enable the report summary (see “The bugs summary” on page 78) you will see the total number of each error at each source line.

## Limiting the number of errors

If your program is generating too many errors for convenient analysis, you can arrange for it to exit (with a non-zero exit code) after displaying a certain number of errors by adding the line

```
insure++.exit_on_error number
```

to your `.psrc` file and re-running the program. After the indicated number of errors, the program will exit. If number is less than or equal to zero, all errors are displayed.

## Changing stack traces

There are two potential modifications you can make to alter the appearance of the stack tracing information presented by Insure++ to indicate the location of an error.

By default, Insure++ will read your program's symbol table at start-up time to get enough information to generate stack traces. To get file and line information, you will need to compile your programs with debugging information turned on (typically via the `-g` switch). If this is a problem, Insure++ can generate its own stack traces for files compiled with Insure++. You can select this mode by adding the options

```
insure++.symbol_table off
insure++.stack_internal on
```

to your `.psrc` file. The `stack_internal` option will take effect after you recompile your program, while the `symbol_table` option can be toggled at runtime. In this case, the stack trace will display

```
** routines not compiled with insure **
```

in place of the stack trace for routines which were not compiled with Insure++. This will also make your program run faster, particularly at start-up, since the symbol table will not be read.

If your program has routines which are deeply nested, you may see very long stack traces. You can reduce the amount of stack tracing information made available by adding an option like

```
insure++.stack_limit 4
```

into your `.psrc` file. If you run your program again, you will see at most 1 the last four levels of the stack trace with each error.

The value "0" is valid and effectively disables tracing.

The value "-1" is the default and indicates that the full stack trace should be displayed, regardless of length.

Stack traces are also presented to show the function calling sequence when blocks of dynamically allocated memory were allocated and freed. In a manner similar to the `stack_limit` option, the `malloc_trace` and `free_trace` options control how extensive these stack traces are.

## Searching for source code

Normally, Insure++ remembers the directory in which each source file was compiled and looks there when trying to display lines of source code in error messages. Occasionally your source code will no longer exist in this directory, possibly because of some sophisticated "build" or "make" process.

You can give Insure++ an alternative list of directories to search for source code by adding a value such as

```
source_path .;c:\users\boswell\src;c:\src
```

to the `.psrc` file in your current working or `HOME` directories.

The list can contain any number of directories separated by semicolons (`;`).

**Note:** Insure++'s error messages normally indicate the line of source code responsible for a problem on the second line of an error report, after the `>>` mark. If this line is missing from the report, it means that the source code could not be found at runtime.

## Suppressing error messages

The previous sections described issues which can affect the appearance of particular error messages. Another alternative is to completely suppress error messages of a given type which you either cannot or do not want to correct.

The simplest way of achieving this is to add lines similar to

```
insure++.suppress EXPR_NULL, PARM_DANGLING
```

to your `.psrc` file and re-run the program. No suppressed error messages will be displayed, although they will still be counted and displayed in the report summary (see “The bugs summary” on page 78).

In this context, certain wild-cards can be applied so that, for instance, you can suppress all memory leak messages with the command

```
insure++.suppress LEAK_*
```

You can suppress all errors with the command

```
insure++.suppress *
```

which has the effect of only creating an error summary. If the error code has sub-categories, you can disable them explicitly by listing the sub-category codes in parentheses after the name, e.g.,

```
insure++.suppress BAD_FORMAT(sign, compatible)
```

Alternatively,

```
insure++.suppress BAD_FORMAT
```

suppresses all sub-categories of the specified error class.

## Suppressing error messages by context

In addition to suppressing and unsuppressing errors by category or file, you can also suppress and unsuppress error messages by context. For example, to suppress `READ_NULL` errors occurring in routines with names beginning with the characters `sub`, enter:

```
insure++.suppress READ_NULL { sub* * }
```

The interpretation of this syntax is as follows:

- The stack context is enclosed by a pair of braces.
- Routine names can either appear in full or can contain the `*` or `?` wildcard characters. The former matches any string, while the latter matches any single character.
- An entry consisting of a single `*` character matches any number of functions, with any names.
- Entries in the stack context are read from left to right with the left-most entries appearing lowest (or most recently) in the call stack.

With these rules in mind, the previous entry is read as

- The lowest function in the stack trace (i.e., the function generating the error message) must have a name that begins with `sub` followed by any number of other characters.
- Any number of functions of any name may appear higher in the function call stack.

A rather drastic, but common, action is to suppress any errors generated from within calls to the X Window System libraries. If we assume that these functions have names which begin with either "X" or "\_X", we could achieve this goal with the statements

```
insure++.suppress all { * X* * }
insure++.suppress all { * _X* * }
```

which suppresses errors in any function (or its descendents) which begins with either of the two sequences.

As a final example, consider a case in which we are only interested in errors generated from the routine `foobar` or its descendents. In this case, we can combine `suppress` and `unsuppress` commands as follows

```
insure++.suppress all
insure++.unsuppress all { * foobar * }
```

## Suppressing Messages by File/Line

In addition to suppressions based on stack traces, you can suppress error messages based on the file/line generating the message.

The syntax for this type of suppression is:

```
file:line#
in file
```

Examples:

```
suppress readbadindex at foo.h:32
```

This suppresses `readbadindex` error messages at line 32 of `foo.h` at both compile-time and run-time.

```
suppress parserwarning in header.h
```

This suppresses all parser warnings in `header.h`.

Wildcards are not supported in filenames for this syntax. However, this syntax can be used at both compile-time and run-time (unlike stack trace suppressions, which can only be used at runtime).

It is illegal to have both a stack trace suppression and a file/line suppression on the same line, e.g.:

```
suppress myerror {a b c} at foo.c:3
```

## Suppressing C++ warning messages

The warning messages that Insure++ displays during parsing of C++ code (see “C++ compile time warnings” on page 50) can be suppressed if the user does not wish to correct the code immediately. For example, to suppress the warning from that section, simply add

```
insure++.suppress_warning 13-2
```

to your `.psrc` file and recompile. The warning messages will no longer be displayed.

## Suppressing other warning messages

For other compile time warning messages that do not have an associated number, there is another suppress option available. The `suppress_output` option takes a string as an argument and will suppress any message that includes text which matches the string. For example, the option

```
insure++.suppress_output wrong arguments passed
```

would suppress the warning from the previous section, as well as any others that included this text string.

## Enabling error messages

Normally, you will be most interested in suppressing error messages about which you can or want to do nothing. Occasionally, you will want to enable one of the options that is currently suppressed, either by system default (See “Error Codes” on page 199) or one of your own `.psrc` files. This is achieved by adding a line similar to the following to your `.psrc` file.

```
insure++.unsuppress RETURN_FAILURE
```

## Stretchy arrays

Another problem that comes up infrequently but causes problems is “stretchy” arrays. Many programmers build structures in which the last element is an array whose size is only determined at runtime. Consider the following code

```
1:      /*
2:      * File: stretch1.c
3:      */
4:      #include <stdlib.h>
5:
6:      struct stretchy {
7:          int nvals;
8:          int data[1];
9:      };
10:
11:     struct stretchy *create(nvals)
12:         int nvals;
13:     {
14:         int size;
15:
16:         size = sizeof(struct stretchy) +
17:              (nvals-1)*sizeof(int);
18:         return (struct stretchy *)malloc(size);
19:     }
20:
21:     main()
22:     {
23:         struct stretchy *s;
```

```

24:             int i;
25:
26:             s = create(10);
27:             for(i=0; i<10; i++) s->data[i] = 0;
28:         }

```

Because the memory allocation in line 18 takes into account the extra memory required for the ten elements in the array, the loop in line 27 is actually valid. Insure++ can automatically detect possible stretchy arrays and treats them accordingly. The `autoexpand` Advanced Option controls this feature. For complete details on this option, see “Options used by Insure++” on page 179.

An interesting exercise is to change the loop in line 27 of the above code to

```
for(i=0; i<=10; i++) s->data[i] = 0;
```

Insure++ catches this. The above change is provided to you as example `stretch2.c`.

**Note:** Elements of anonymous unions and structures (i.e. unions and structures without a tag) cannot be marked as stretchy, as there is no way to identify them to Insure++. If you have a stretchy array in such a union or structure, you will need to edit your source code to insert a tag if you want to declare the array stretchy.

## Report summaries

Normally, you will see error messages for individual errors as your program proceeds. Using the other options described so far, you can enable or disable these errors or control the exact number seen at each source line. This technique is most often used to systematically track down each problem, one by one.

However, it is often useful to obtain a summary of the problems remaining in a piece of code in order to track its progress. Insure++ supports the following types of summary reports.

- A bug summary which lists all outstanding bugs according to their error codes.
- A leak summary which lists all memory leaks - i.e., places where memory is being permanently lost.

- An outstanding summary which lists all outstanding memory blocks - i.e., places where memory is not being freed, but is not leaked because a valid pointer to the block still exists.
- A coverage summary which indicates how much of the application's code has been executed.

None of these is displayed by default.

## The bugs summary

This report summary is enabled by adding the option

`insure++.summarize bugs`

to your `.psrc` file and re-running your program.

In addition to the normal error reports, you will then also see a summary such as the one shown below.

```
***** INSURE SUMMARY ***** v6.0 **
      Program                : gs
      Arguments               : golfer.ps
      Directory                : C:\src\trf\gsb
      Compiled on              : Nov 5, 2000 15:40:37
      Run on                   : Nov 5, 2000 15:44:29
      Elapsed time             : 00:01:06
*****
```

PROBLEM SUMMARY - by type  
=====

Problem	Detected	Suppressed
EXPR_BAD_RANGE	7	0
READ_UNINIT_MEM	23	0
BAD_DECL	1	0
TOTAL	31	0

PROBLEM SUMMARY - by location  
=====

```
EXPR_BAD_RANGE: Expression exceeded range, 7 occurrences
                5 at ialloc.c, 170
```

```

    1 at ialloc.c, 176
    1 at ialloc.c, 182

READ_UNINIT_MEM: Reading uninitialized memory, 23 occurrences
    7 at gxcpath.c, 137
    7 at gxcpath.c, 241
    1 at gdevx.c, 424
    1 at gdevxini.c, 213
    2 at gdevxini.c, 221
    1 at gdevxini.c, 358
    1 at gdevxini.c, 359
    1 at gdevxini.c, 422
    1 at gdevxini.c, 454
    1 at gdevxini.c, 514

BAD_DECL: Global declarations are inconsistent, 1 occurrence
    1 at gdevx.c, 93

```

The first section is a header which indicates the following information about the program being executed.

- The name of the program.
- Its command line arguments, if available.
- The directory from which the program was run.
- The time it was compiled.
- The time it was executed.
- The length of time to execute.

This information is provided so that test runs can be compared accurately as to the arguments and directory of test. The time and date information is supplied to correlate with bug tracking software.

The second section gives a summary of problems detected according to the error code and frequency. The first numeric column indicates the number of errors detected but not suppressed. This is the total number of errors, which might differ from the number reported, since, by default, only the first error of any particular type is reported at each source line. The second column indicates the number of bugs which were not displayed at all due to `suppress` commands.

The third section gives details of the information presented in the second section, broken down into source files and line numbers.

## The leak summaries

The simplest memory leak summary is enabled by adding the line

```
insure++.summarize leaks outstanding
```

to your `.psrc` file and re-running your program.

The output indicates the memory (mis)use of the program, as shown below.

```
***** INSURE SUMMARY ***** v6.0 **
      Program                : leak
      Arguments              :
      Directory              : C:\whicken\test
      Compiled on            : Nov 5, 2000 15:09:05
      Run on                 : Nov 5, 2000 15:09:31
      Elapsed time          : 00:00:02
*****
MEMORY LEAK SUMMARY
=====
```

4 outstanding memory references for 45 bytes.

Leaks detected during execution

```
-----
      10 bytes      1 chunk      allocated at leak.c, 6
```

Leaks detected at exit

```
-----
      10 bytes      2 chunk      allocated at leak.c, 7
```

Outstanding allocated memory

```
-----
      15 bytes      1 chunk      allocated at leak.c, 8
```

The first section summarizes the “memory leaks” which were detected during program execution, while the second lists leaked blocks that were detected at program exit. These are potentially serious errors, in that they typically represent continuously increasing use of system resources. If the program is “leaking” memory, it is likely to eventually exhaust the system resources and will probably crash. The first number displayed is the total amount of memory lost at the indicated source line, and the second is the number of chunks of memory lost. Note that multiple chunks *of different sizes* may be lost at the same source line - depending on which options

you are using. To customize the report, there are three Advanced Options available: `leak_combine`, `leak_sort`, and `leak_trace`.

The `leak_combine` option controls how Insure++ merges information about multiple blocks. The default behavior is to combine all information about leaks which were allocated from locations with identical stack traces (`leak_combine trace`). It may be that you would rather combine all leaks based only on the file and line they were allocated, independent of the stack trace leading to that allocation. In that case, you would use `leak_combine location`. Or, you may simply want one entry for each leak (`leak_combine none`).

The `leak_sort` option controls how the leaks are sorted after having been combined. The options are `none`, `location`, `trace`, `size`, and `frequency` (`size` is the default). Sorting by `size` lets you look at the biggest sources of leaks, sorting by `frequency` lets you look at the most often occurring source of leaks, and sorting by `location` provides an easy way to examine *all* your leaks.

The `leak_trace` option causes a full stack trace of each allocation to be printed, in addition to the actual file and line where the allocation occurred.

The third section shows the blocks which are allocated to the program at its termination and which have valid pointers to them. Since the pointers allow the blocks to still be freed by the program (even though they are not), these blocks are not actually leaked. This section is only displayed if the `outstanding` keyword is used. Normally, these blocks do not cause problems, since the operating system will reclaim them when the program terminates. However, if your program is intended to run for extended periods, these blocks are potentially more serious.

## The coverage summary

The coverage summary is enabled by adding the line

```
insure++.summarize coverage
```

to your `.psrc` file and re-running your program.

In addition to the normal error reports, you will see a summary indicating how much of the application's source code has been tested. The exact

form of the output is controlled by the `.psrc` file option `coverage_switches`, which specifies the command line switches passed to the `tca` command to create the output.

If this variable is not set, it defaults to

```
insure++.coverage_switches -dS
```

which displays an application level summary of the test coverage such as

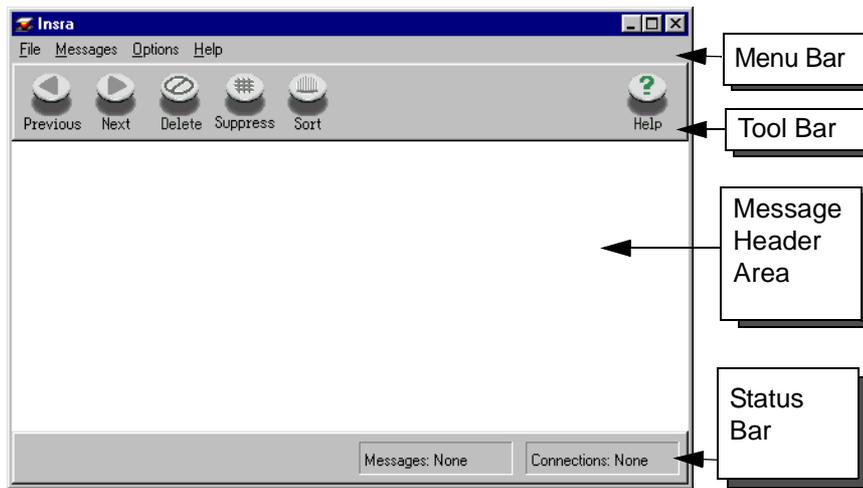
```
COVERAGE SUMMARY
=====
11      blocks untested
42      blocks tested

78% covered
```



# Insra

Insra is a graphical user interface for displaying error messages generated by Insure++. The messages are summarized in a convenient display, which allows you to quickly navigate through the list of bug reports and violation messages, suppress messages, invoke an editor for immediate corrections to the source code, and delete messages as bugs are fixed.



## The Insra display

### Status bar

During compilation/runtime, Insure++ makes a connection to Insra each time an error is detected. The status bar reports the number of error messages currently displayed and the number of active connections. An active connection is denoted by a yellow star to the left of the session header. A connection remains active as long as the program is compiling/running. Insra will not allow you to delete a session header as long as its

connection remains active, and you may not exit Insra until all connections have been closed.

## Tool bar

The tool bar allows you to:

- Scroll up and down through messages.
- Delete selected messages as bugs are fixed.
- Suppress errors detected by Insure++.
- Sort messages by order (time) reported, error category, or directory and file.
- Kill the selected active connection.

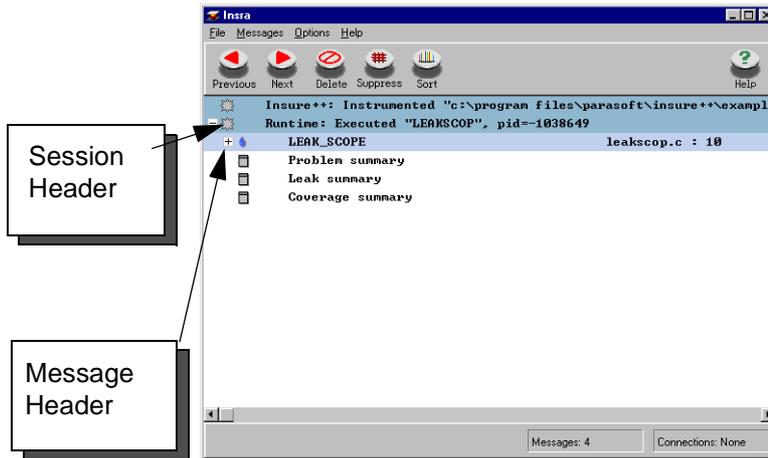
## Message header area

The message header area contains session headers and message headers for programs currently connected to Insra.

## Session header

When the first error is detected for a particular compilation or execution, a session header is sent to Insra. The session header includes the following information:

- Compilation/execution
- Source file/program
- Host on which the process is running
- Process ID



## Message header

There are several types of message headers. Messages generated by Insure++ include:

- Error Category, e.g. LEAK\_SCOPE
- File name
- Line number

Message headers will also appear for various summary reports generated by Insure++. These reports are generated using the `.psrc` options (see “Report summaries” on page 77). Double-clicking on a message header will open up the message window for the error or summary report selected.

## Message window

The message window opens when you double-click on a message header. This window contains the error message or summary report for the selected message header.

## Error message

Insure++ error messages include:

- Line of source code where the error occurred.
- Explanation of the error detected.
- Stack traces for quick reference to the original source.

The stack traces are “live” and can be clicked once to launch an editor for viewing and correcting the indicated line of code. For more information, see “Viewing source files” on page 92.

All messages sent to Insra are marked with a special icon. Please refer to the following table for a brief description of each icon.

Icon	Explanation
	Insure++ error message
	Insure++ summary report
	Memory leak
	Caught exception

## Sending messages to Insra

By default, all Insure++ output is sent to stderr. To redirect messages to Insra, simply add the following line to your `.psrc` file.

```
insure++.report_file insra
```

This will redirect both compile-time and run-time messages to Insra.

The option

```
insure++.runtime.report_file insra
```

will send only runtime messages to Insra. Compile-time messages will continue to be sent to `stderr`.

The option

```
insure++.compile.report_file insra
```

will send only compile-time messages to Insra. Runtime messages will continue to be sent to `stderr`.

With `insure++.report_file insra` in your `.psrc` file, each time an error is detected, Insure++ attempts to establish a connection to Insra. If Insra is not yet running, it will automatically start. Once the connection is established, a session header and all corresponding message headers will be reported in the order they were detected. Each new compilation or program, with its own session header and messages, will be displayed in the order in which it connected to Insra.

## Viewing and navigating

Message headers sent to Insra are denoted by a specific icon (see “The Insra display” on page 84). The body of the currently selected message is displayed in a separate message window. Double-click the message header to view the message itself. The message header area and the message window are both resizable, and scroll bars are also available to access text that is not visible.

Currently active messages become inactive when they are deleted or suppressed.

## Selecting an editor

In addition to the location of the source file, Insra must also know the name of your editor and the command line syntax in order to display the correct file and line from the original source code.

Insra obtains this information by reading the `.psrc` option value

```
insra.visual [editor_command]
```

This value may contain the special tokens `%f` and `%l`, which represent the file name and line number, respectively.

The command will then be executed to load the file into your editor. It is most important to include the full path of any binary that lives in a location not pointed to by your `PATH` environment variable. If the variable has not been set, `vi` will be used by default.

Some editors are not X applications and must be run in a terminal window. `vi` requires the following command in order to lead the file successfully:

```
insra.visual xterm -e vi +%l %f
```

Other editors (e.g. Emacs) do not require an external terminal program like `xterm` when configured for use as an X application. In this case, the command string should be similar to the following:

```
insra.visual emacs +%l %f
```

**Note:** Most implementations of `vi` and Emacs appear to be sensitive to the order of the line number and file name command line arguments, requiring the line number to precede the file name.

## Deleting messages

Once error messages have been read and analyzed, the user may wish to clear them from the window. The **Delete** button on the Insra toolbar allows you to remove error messages from the display as errors are corrected in your code. A message or an entire session may be removed from the display by selecting an entry in the message header area and clicking the **Delete** button. A message can also be deleted by selecting **Messages/Delete** from the menu bar.

## Suppressing messages

You can easily suppress (turn "off") error messages which you do not want Insure++ to generate. This window allows you to insert, modify, and delete suppression options for Insure++ error messages. The suppression options you choose can be saved into your `.psrc` files so that they will be used again the next time you use Insure++.

## The Toolbar

Moving from left to right across the toolbar:

The **Previous** and **Next** arrows select the next higher or next lower suppression option, respectively.

The **Up** and **Down** arrows move the currently selected suppression option up or down in the order of options. Because Insure++ follows suppression options from top to bottom, the order in which they are listed affects the outcome of the suppression. (For more information, see “Reports” on page 67.)

The **Delete** button deletes the currently selected suppression option.

The **Insert** button inserts a new suppression option below the currently selected option. If you had an error message selected when you pressed the **Suppress** button on the Insra GUI, a suppression option for that particular error message will be inserted. Otherwise, the default suppression option (suppress all error messages) will be inserted. Suppression options are easy to edit. To change an option, simply follow the directions given below.

The **Save** button writes all the suppression options which have been marked as persistent (see below) into the indicated `.psrc` files (see below). These suppression options will be in effect the next time you use Insure++.

The **Help** button provides context-sensitive help, which in this window means that clicking anywhere will bring up this file.

The **Close** button closes the suppression window.

## Editing suppression options

An individual suppression option consists of five parts, listed below from left to right:

- **Suppress/Unsuppress**: This field specifies whether the error message listed is to be suppressed (as indicated by a speaker with an X through it) or unsuppressed (indicated by a speaker with no X through it). Double-clicking on the field toggles between suppressing and unsuppressing the error message.

- **Persistence:** This field specifies whether the suppression option will be saved to the `.psrc` file under which it is listed. Double-clicking this field toggles it from persistent (the field is checked) to temporary (the field is unchecked). Options marked as persistent will be added to the appropriate `.psrc` files when the save button is clicked. Options marked as temporary will be discarded. Options with an X in this field cannot be made persistent, either because they are hardwired or because the file in which it is placed is not writable. New options are marked as persistent by default.
- **Item:** Double-clicking this field allows you to type in the name of the error message you would like to suppress or unsuppress. You can use a wildcard (\*) to match all error messages and also suppress and unsuppress by error category and context. For more information on suppressing error messages, see “Suppressing error messages” on page 73.
- **File:** Double-clicking this field allows you to type in the file for which you would like to suppress or unsuppress messages. Entering a blank field will insert a \* which will match all files.
- **Note:** You may use this field to enter your own notes regarding the suppression option listed.

## Configuration (.psrc) files

The headers in the window show the various locations in which `.psrc` files reside. Insra will display the suppression options as read from each file under the appropriate header. When you add a new option using the Insert button, it will be inserted below the currently selected option. You can then move it into the file in which you would like it saved, or mark it as temporary by double-clicking the persistence field (see above). There are two special locations where suppression options may reside other than actual `.psrc` files: hardwired options and command line options. The former are set internally by Insure++, and therefore cannot be permanently changed. They can be edited and/or removed in the window temporarily, however. The latter are options passed using the `-Zop` and `-Zoi` options on the Insure++ command line. These options, like hardwired options, cannot be made persistent, but can be moved into a `.psrc` file if you decide that you want to make them permanent.

## Kill process

When an active connection is selected, pressing the **Kill** button will stop the selected compilation or execution.

## Viewing source files

You can view the corresponding source file and line number for a particular error message by double clicking any line of the stack trace displayed in the message window. In most cases, the file and line number associated with a given message have been transmitted to Insra. If Insra is unable to locate the source file, a dialog box will appear requesting that you indicate the correct source file.

## Saving/loading messages to a file

All current messages can be saved to a file by selecting **File/Save** or **File/Save As** from the menu bar. A dialog box allows you to select the destination directory and name of the report file. Report files have the default extension `rpt`. After a report file name has been selected, subsequent **File/Save** selections save all current messages into the report file without prompting for a new filename. A previously saved report file can be loaded by selecting **File/Load** from the menu bar. A dialog box then allows you to select which report file to load.

## Help

On-line help can be obtained by choosing **Help** from the menu bar. This provides a list of topics on the use of Insra.

## Setting preferences

You can modify Insra's appearance with `.psrc` configuration options. These options are:

`insra.body_background_color` [color]

Specifies the color used for the message body area background. The default is white.

`insra.body_font` [font]

Specifies the font used for the message body text. The default is fixed.

`insra.body_height` [number of rows]

Specifies the starting height of the message window in number of rows of visible text. The default is 8.

`insra.body_text_color` [color]

Specifies the color used for the message body text. The default is black.

`insra.body_width` [columns of text]

Specifies the starting width of the message window in number of columns of visible text. The default is 80, but if this value is set to a different value than `header_width`, then the larger value will be used.

`insra.button_style` [round|square]

Specifies the shape of buttons that will be shown on toolbars. The default is round.

`insra.coloured_shadows` [on|off]

Specifies if round button shadows will be re-colored with the color of the application background. The default is on.

`insra.expose_on_message` [on|off]

Specifies if the Insra GUI will be placed on top of windows stack if it receives a new message. The default is off. (This option works only in by-time view mode.)

`insra.follow_messages` [on|off]

Specifies if the main window messages area will be automatically scrolled to follow arriving messages. The default is off. Note: this option works only in by-time view mode.

`insra.header_background_color` [color]

Specifies the color used for the message header area background. The default color is white.

`insra.header_font` [font]

Specifies the font used for the message header text. The default is fixed.

`insra.header_height [number of rows]`  
Specifies the starting height of the message header in number of rows of visible text. The default is 8.

`insra.header_highlight_color [color]`  
Specifies the color used to indicate the currently selected message or session header in the message header area. The default is `LightSteelBlue2`.

`insra.header_highlight_text_color [color]`  
Specifies the color used for the text of the currently selected message of session header in the message header area. The default is black.

`insra.header_session_color [color]`  
Specifies the color used for session header text. The default is `LightSkyBlue3`.

`insra.header_session_text_color [color]`  
Specifies the color used for session header text. The default is black.

`insra.header_text_color [color]`  
Specifies the color used for message header text. The default color is black.

`insra.header_width [number of columns]`  
Specifies the starting width of the header area in number of columns of visible text. The default is 80, but if this value is set to a different value than `body_width`, the larger value will be used.

`insra.mark_unique [on|off]`  
Specifies if messages not duplicated across all connections from the same tool has to be marked by special arrow-like icon. The default is on.

`insra.port [port_number]`  
Specifies which port Insra should use to communicate with Insure++ compiled programs. The default is 3255.

`insra.sourcepath [dir_path1 dir_path2 ...]`  
Specifies directories to be searched by Insra to find source files launching editor or showing source lines.

`insra.toolbar [on or off]`  
Specifies whether Insra's toolbar is displayed. All toolbar commands can be chosen from the menu bar. The default is on.

```
insra.viewmode [by_error|by_file|by_time|off|tool]
```

Specifies initial view mode. Allows to override view mode settings made by tools connecting with Insra.

by\_error - Insra GUI is initially set in view-by-error-category mode

by\_file - Insra GUI is initially set in view-by-file mode

by\_time - Insra GUI is initially set in view-by-time mode

off - Insra GUI is launched with the default view mode (i.e. by-time mode)

tool - means that view mode will be set by tool that first connects with the Insra GUI. This is the default setting.

```
insra.visual [editor command]
```

Specifies how Insra should call an editor to display the line of source code causing the error. Insra will match the %l token to the line number and the %f token to the file name before executing the command. Setting this option with no command string disables source browsing from Insra. The default is

```
xterm -e vi +%l %f
```

## Troubleshooting

### Insra does not start automatically

Symptom: While compiling or running, your program seems to hang when error output is directed to Insra and Insra is not yet running.

Solution: Run Insra by hand. Type

```
insra &
```

at the prompt, wait for the Insra window to appear and then run or compile your program again. Output should now be sent to Insra.

### Multiple users of Insra on one machine

Symptom: When more than one user is attempting to send message reports to Insra, messages are lost.

Solution: Each invocation of Insra requires a unique port number. By default, Insra uses port 3255. If collisions are experienced, e.g. multiple users are on one machine, set the .psrc option insra.port to a different

port above 1024. Ports less than 1024 are officially reserved for suid-root programs and should not be used with Insra.

## Source browsing is not working

Symptom:

```
***Error while attempting to spawn browser execvp failed!
```

Solution: Insra attempted to launch your editor to view the selected source file, but could not locate your editor on your path. Please make sure that this application is in a directory that is on your path or that you call with its complete pathname.

# Selective Checking

By default, Insure++ will check for bugs for the entire duration of your program. If you are only interested in a portion of your code, you can make some simple, unobtrusive changes to the original source to achieve this.

When you compile with `insure`, the pre-processor symbol `__INSURE__` is automatically defined. This allows you to conditionally insert calls to enable and disable runtime checks.

For example, if you are not interested in events occurring during the execution of a hypothetical function `grind_away`. To disable checking during this function, you can modify the code as shown below.

```
grind_away() {
#ifdef __INSURE__
    _Insure_set_option("runtime", "off");
#endif
    ... code ...
#ifdef __INSURE__
    _Insure_set_option("runtime", "on");
#endif
}
```

Now when you compile and run your program, it will not check for bugs between the calls to `_Insure_set_option`.

If you do not want to modify the code for the `grind_away` function itself, you can add calls to `_Insure_set_option` around the calls to `grind_away`.

# Interacting with Debuggers

While it is our intent that the error messages generated by Insure++ will be sufficient to identify most programming problems, it will sometimes be useful to have direct access to the information known to Insure++. This can be useful in the following situations

- You are running your program from a debugger and would like to cause a breakpoint whenever Insure++ discovers a problem.
- You are tracing an error using the debugger and would like to monitor what Insure++ knows about your code.
- You wish to add calls to your program to periodically check the status of some data.

## Available functions

Whenever Insure++ detects an error, it prints a diagnostic message and then calls the routine `_Insure_trap_error`. This is a good place to insert a breakpoint if you are working with a debugger.

The following functions show the current status of memory and can be called either from your program or the debugger. Remember to add prototypes for the functions you use, particularly if you are calling these C functions from C++ code.

```
int _Insure_mem_info(void *pmem);
```

Displays information that is known about the block of memory at address `pmem`. (Returns zero.)

```
int _Insure_ptr_info(void **pptr);
```

Displays information about the pointer at the indicated address. (Returns zero.)

The following function lists all currently allocated memory blocks, including the line number at which they were allocated. It can be called directly from your program or from the debugger.

```
long _Insure_list_allocated_memory(int mode);
```

0 - Just the total allocation

- 1 - "Newly-Allocated" or reallocated blocks
- 2 - Everything

## Sample debugging session

The use of these functions is best illustrated by example.

Consider the following program

```

1:      /*
2:      * File: bugsfunc.c
3:      */
4:      #include <stdlib.h>
5:
6:      main()
7:      {
8:          char *p, *q;
9:
10:         p = (char *)malloc(100);
11:
12:         q = "testing";
13:         while(*q) *p++ = *q++;
14:
15:         free(p);
16:     }
```

Compile this code under Insure++ in the normal manner (with the `-Zi` option), and start the debugger in the normal manner.

**Note:** The instructions shown here assume that the debugger you are using is similar to `dbx`. If you are using another debugger, similar commands should be available.

```

$ dbx bugsfunc
Reading symbolic information...
Read 4650 symbols
(dbx)
```

If the debugger has trouble recognizing and reading the source file, you may need to use the `rename_files` or `.psrc` option. See "Configuration Options" on page 175 for more information about this option.

It is generally useful to put a breakpoint in `_Insure_trap_error` so that you can get control of the program whenever an error occurs. In this case, we run the program to the error location with the following result

```
(dbx) stop in _Insure_trap_error
(2) stop in _Insure_trap_error
```

**Note:** The above may not work if you have linked against the shared Insure++ libraries (the default). If you cannot set a breakpoint as shown above, it is because the shared libraries are not loaded by the debugger until the program begins to run. You can avoid this problem by linking against the static Insure++ libraries (see the `static_linking` option in the section “Configuration Options” on page 175) or by setting a breakpoint in `main` and starting the program before setting the breakpoint on `_Insure_trap_error`.

```
(dbx) run
Running: bugsfunc
[bugsfunc.c:15] **FREE_BODY**
>>          free(p);

Freeing memory block from body: p

Pointer : 0x0001e26f
Stack trace where the error occurred:
          main() bugsfunc.c, 15

**Memory corrupted. Program may crash!!**
stopped in _Insure_trap_error at line ...
          217      return;

(dbx)
```

The program is attempting to free a block of memory by passing a pointer that doesn’t indicate the start of an allocated block. The error message shown by Insure++ identifies the location at which the block was allocated and also shows us that the variable `p` has been changed to point into the middle of the block, but it doesn’t tell us where the value of `p` changed.

We can use the Insure++ functions from the debugger to help track this down.

Since the program is already in the debugger, we can simply add a breakpoint back in `main` and restart it

```
(dbx) func main
(dbx) stop at 10
```

```
(4) stop at "examples/bugsfunc.c":10
(dbx) run
Running: bugsfunc
stopped in main at line 10 in file "bugsfunc.c"
    10         p = (char *)malloc(100);
```

To see what is currently known about the pointers `p` and `q`, we can use the `_Insure_ptr_info` function

**Note:** The `_Insure_ptr_info` function expects to be passed the *address* of the pointer, not the pointer itself. To see the contents of the memory indicated by the pointers, use the `_Insight++_mem_info` function.

```
(dbx) print _Insight++_ptr_info(&p)
Uninitialized
(dbx) print _Insight++_ptr_info(&q)
Uninitialized
```

Both pointers are currently uninitialized, as would be expected.

To see something more interesting, we can continue to line 13 and repeat the previous steps.

```
(dbx) stop at 13
(6) stop at "examples/bugsfunc.c":13
(dbx) cont
stopped in main at line 13 in file "bugsfunc.c"
    13         while(*q) *p++ = *q++;
(dbx) print _Insure_ptr_info(&p)
Pointer          : 0x0001e358 (heap)
Offset           : 0 bytes
In Block         : 0x0001e358 thru 0x0001e3bb
                  (100 bytes)
                  p, allocated at bugsfunc.c, 10
```

The variable `p` now points to a block of allocated memory. You can check on all allocated memory by calling `_Insure_list_allocated_memory`.

```
(dbx) print _Insure_list_allocated_memory()
1 allocated memory block, occupying 100 bytes.
[bugsfunc.c:10] 100 bytes.
```

Finally, we check on the second pointer `q`.

```
(dbx) print _Insure_ptr_info(&q)
Pointer          : 0x00016220 (global)
Offset           : 0 bytes
In Block         : 0x00016220 thru 0x0001e3bb
```

```
(100 bytes)
"testing", declared at bugsfunc.c,12
```

Everything seems OK at this point, so we can continue to the point at which the memory is freed and check again.

```
(dbx) next
      15      free(p);
(dbx) print _Insure_ptr_info(&p)
Pointer      : 0x0001e35f (heap)
Offset       : 7 bytes
In Block     : 0x0001e358 thru 0x0001e3bb
              (100 bytes)
              p, allocated at bugsfunc.c, 10
```

The critical information here is that the pointer now points to an offset 7 bytes from the beginning of the allocated block. Executing the next statement, `free(p)`, will now cause the previously shown error, since the pointer doesn't point to the beginning of the allocated block anymore.

Since everything was correct at line 12 and is now broken at line 15, it is simple to find the problem in line 13 in which the pointer `p` is incremented while looping over `q`.



# Tracing

Tracing is a very useful enhancement of Insure++ for C++ programmers. Because C++ is such a complicated language, programmers may never know which functions are being called or in which order. Some functions are called during initialization before the main program begins execution. Tracing provides the programmer with the ability to see how functions, constructors, destructors, and more are called as the program runs.

Insure++ prints a message at the entry to every function which includes the function name, filename, and line number of the command that called it.

A typical line of output from tracing looks like this:

```
function_name filename, line_number
```

By default, the output is indented to show the proper depth of the trace.

## Turning tracing on

By default, tracing is turned off. The easiest way to turn tracing on is to set the `trace on` value in your Advanced Options. This turns on tracing for the entire program. See “Options used by Insure++” on page 179 for more information about this option.

**Note:** To get a full trace, you must use the `-zi` compiler switch on your `insure` compile line. To get file names and line numbers in the trace output, you must use the `stack_internal on` option when compiling your program (see “Options used by Insure++” on page 179).

You may not want to always do this, because your program will slow down while every function call prints information.

This problem can be minimized by selectively turning on tracing during the execution of your program only in those sections of the code where you need it most. This can be done using the special Insure++ command

```
void _Insure_trace_enable(int flag)
```

- `flag = 0` turns tracing off
- `flag = 1` turns tracing on

There is one more special Insure++ function that works with tracing. This function may be used to add your own messages to the trace.

```
void _Insure_trace_annotate(int indent, char *format, ...)
```

- `indent = 0` means string is placed in column zero
- `indent = 1` means string will be indented at proper level
- `format` should be a normal `printf`-style format string

## Directing tracing output to a file

The default output for tracing data, like all other Insure++ output, is set in the **General** tab of the Insure++ Control Panel. You can direct tracing output to a specific file by setting a `trace_file filename` value in your Advanced Options. For more information about this option see “Options used by Insure++” on page 179.

**Note:** When you use this option, Insure++ prints a message reminding you where the tracing data is being written. If you would like to eliminate these reminders, you can use the `trace_banner off` option in your Advanced Options. See “Options used by Insure++” on page 179 for more information.

## Example

The following code can be found in the `examples\cpp` directory as the file `trace.cpp`.

```

1:      /*
2:      * File: trace.cpp
3:      */
4:      int twice(int j) {
5:          return j*2;
6:      }
7:      class Object {
8:      public:
9:          int i;
10:         Object() {
11:             i = 0;
12:         }
13:         Object(int j) {
14:             i = j;
15:         }
16:         operator int() { return twice(i); }
17:     };
18:     int main() {
19:         Object o;
20:         int i;
21:
22:         i = o;
23:         return i;
24:     }

```

If you compile and link `trace.cpp` with the `-zi` option and the `stack_internal` on option (see “Options used by Insure++” on page 179), and then run the executable with the `trace on` value (see “Options used by Insure++” on page 179) set in your Advanced Options, you will see the following output:

```

main
    Object::Object      trace.cpp, 19
    Object::operator inttrace.cpp, 22
                       twice      trace.cpp, 16

```

# Signals

In addition to its other error checks, Insure++ also traps certain signals. It does this by installing handlers when your program starts up. These do not interfere with your program's own use of signals - any code which manipulates signals will simply override the functions installed by Insure++.

## Signal handling actions

When a signal is detected, Insure++ does the following

- Prints an informative error.
- Logs the signal in the Insure++ report file, if one is being used.
- Calls the function `_Insure_trap_error`.
- Takes the appropriate action for the signal.

If this last step will result in the program terminating, Insure++ attempts to close any open files properly. In particular, the Insure++ report file will be closed. Note that this can only work if the program hasn't crashed the I/O system. If, for example, the program has generated a "bus" or similar error, it might not be possible to close the open files. In the worst of all possible scenarios you will simply generate another (fatal) signal when Insure++ attempts to clean up.

## Interrupting long-running jobs

Insure++ installs a handler for the keyboard interrupt command (often **Ctrl-C**, or the **Delete** key). If your program does not override this handler with one of its own, you can abort a long-running program and still get Insure++'s output. If your program has its own handler for this sequence, you can achieve the same effect by adding the following lines to your handler

```
#ifdef _INSURE_
    _Insure_cleanup();
#endif
```

## Which signals are trapped?

By default, Insure++ traps the following signals

```
SIGABRT
SIGBUS
SIGEMT
SIGFPE
SIGILL
SIGINT
SIGIOT
SIGQUIT
SIGSEGV
SIGSYS
SIGTERM
SIGTRAP
```

You can add to or subtract from this list by adding lines to your Advanced Options and re-running the program.

Signals are added to the list with Advanced Option values such as

```
signal_catch SIGSTOP SIGCLD SIGIO
```

and removed with

```
signal_ignore SIGINT SIGQUIT SIGTERM
```

You can omit the `SIG` prefix if you wish.

# Code Insertions

Most programmers write code that makes assumptions about various things that can happen. These assumptions can vary from the very simple, such as “I’m never going to pass a `NULL` pointer to this routine”, to the more subtle, such as “`a` and `b` are going to be positive”.

Whether this is done consciously or not, the problems that occur when these assumptions are violated are often the most difficult to track down. In many cases, the program will run to completion with no indication of error, except that the final answer is incorrect.

## Debugging the hard way

The simple case just described, the `NULL` pointer, will probably be tracked down pretty easily, since `Insure++` will pinpoint the error immediately. A few minutes of work should eliminate this problem.

The second case is much harder.

One option is to add large chunks of debugging code to your application to check for the various cases that you don’t expect to show up. Of course, you normally have an idea of where the problem is, so you start by putting checks there. You then run the code and sort through the mass of output, trying to see where things started to go awry. If you guessed wrong, you insert more checks in other places of the code and repeat the entire process.

If you are lucky, the code you insert to catch the problem won’t add bugs of its own.

Once you’ve found the problem, you can either remove the debugging code (introducing the possibility of deleting the wrong things and bringing in new bugs) or comment it out for use next time (cluttering the source code).

## An easier solution

A second option is to have Insure++ add the checking code to your application automatically and invisibly.

The basic idea is that you tell Insure++ what you'd like to check by providing an "Insure++ interface module". This can be kept separate from your main application and added and removed at compile time.

Furthermore, Insure++ automatically inserts it in every place that you use a particular piece of code so you only have to go through this process once. Finally, errors that are detected are diagnosed in the same way as all other Insure++ errors. You get a complete report of the source file, line number, and function call stack together with any other information that you think is useful.

## An example

Assume that you have a routine in your program called `cruncher` which takes three double precision arguments and returns one. For some reason, possibly connected with the details of your application, you expect the following rules to be true when calls are made to this routine

- The sum of the three parameters is less than 10.
- The first parameter is always greater than zero.
- The return value is never zero.

To enforce these rules with Insure++, you create a file containing the following code.

```

1:      /*
2:      * crun_iic.c
3:      */
4:      (double a, double b, double c)
5:      {
6:          double ret;
7:
8:          if(a+b+c >= 10.) {
9:              iic_error(USER_ERROR,
10:                 "Sum exceeds 10: %f+%f+%f\n",
11:                    a, b, c);
12:          }
13:          if(a <= 0) {
```

```

14:             iic_error(USER_ERROR,
15:                 "a is negative: %f\n", a);
16:         }
17:         ret = cruncher(a, b, c);
18:         if(ret == 0) {
19:             iic_error(USER_ERROR,
20:                 "Return zero: %f,%f,%f => %f\n",
21:                 a, b, c, ret);
22:         }
23:         return ret;
24:     }

```

Note that this looks just like normal C code with the rather strange exception that the routine `cruncher` calls itself at line 17!

This is not normal C code - it's an Insure++ interface description, and it behaves rather like a complicated macro insertion. Wherever your original source code contains calls to the function `cruncher`, they will be replaced by this set of error checks, and the indicated call to the routine `cruncher`.

The net effect will be as though you had added all this complex error checking and printing code manually, except that Insure++ does it automatically for you. Another advantage is the use of the `iic_error` routine rather than a conventional call to `printf` or `fprintf`. The `iic_error` routine performs the same task - printing data and strings, but also includes in its output information about the source file and line number at which the call is being made and a full stack trace.

## Using the interface

Once you've written this interface description, using it is trivial. First, you compile it with the special Insure++ interface compiler, `iic`. If you put the code in a file called `crun_iic.c`, for example, you would type the command

```
iic crun_iic.c
```

This creates a file called `crun_iic.tqs`, which is an "Insure++ interface module".

You can use this module in one of two ways. If you plan to use this interface check on a regular basis during the development of your project, you should insert the value

```
interface_library crun_iic.tqs
```

in your Advanced Options. All future invocations of *Insure++* will then insert this interface check.

If you wish to use the interface check intermittently on some of your compiles, you can add the name of the interface module to the `Insure` command line when you compile and link your source code. For example the command

```
insure -c myfile1.c
```

would become

```
insure crun_iic.tqs -c myfile1.c
```

Note that you can specify more than one interface in any interface file or include multiple interface modules on the `interface_library` line in your Advanced Options or on the `insure` command line.

## Conclusions

This section has shown how you can add your own error checking either to extend or replace that done automatically by *Insure++* by defining “interface modules.” These are actually a very powerful way of extending the capabilities of *Insure++*, and are described more fully in the next section. The current discussion, however, has shown their simplest use.

# Interfaces

The section “Code Insertions” on page 109, described a way of using Insure++ interface descriptions to add user-level checking to function calls. This usage is only one of the things that interfaces can do to extend the capabilities of Insure++. This section describes the purpose of these interfaces in more detail and also shows you how to write your own.

## What are interfaces for?

Interface descriptions provide an extremely powerful facility which allows you to perform extensive checking on functions in system or third party libraries *before* problems cause them to crash.

Most problems encountered in libraries are due to their being called incorrectly from the user application. Insure++ interfaces are designed to trap and diagnose errors where your code makes calls to these functions. This provides the most useful information for correcting the error.

Essentially, an interface is a means of enforcing rules on the way that a function can be called and the side effects it has on memory. Typically, interfaces check that all parameters are of the correct type, that pointers point to memory blocks of the appropriate size, and that parameter values are in correct ranges. Whenever a function is expected to create or delete a block of dynamic memory, interfaces also make calls that allow Insure++’s runtime library to update its internal records.

Writing interfaces for your libraries is a fairly simple task once the basic principles are understood. To help in relating the purpose of an interface to its implementation, the following sections describe two simple examples, one in C and one in C++.

## A C example

Consider the following code, which makes a call to a hypothetical library function `mymalloc`. See the file `mymal.c` below for a definition of `mymalloc`.

```
1:          /*
2:          * File: mymaluse.c
```

```

3:      */
4:      main()
5:      {
6:          char *p, *mymalloc();
7:
8:          p = mymalloc(10);
9:          *p = 0;
10:         return (0);
11:     }

```

In order to get the best from Insure++, you need to summarize the expected behavior of the `mymalloc` function. For this example, let us assume that we want to enforce the following rules:

- The single argument is an integer that must be positive.
- The return value is a pointer to a block of memory that is allocated by the routine.
- The size of the allocated block is equal to the supplied argument.

To do so, we create a file with the following interface.

```

1:      /*
2:      * File: mymal_i.c
3:      */
4:      char *mymalloc(int n)
5:      {
6:          char *retp;
7:          if(n <= 0)
8:              iic_error(USER_ERROR,
9:                  "Negative argument:
%d\n",n);
10:         retp = mymalloc(n);
11:         if(retp) iic_alloc(retp, n);
12:         return retp;
13:     }

```

The key features of this code are as follows

Line 4: A standard ANSI function declaration for the function to be described, including its return type and arguments. (Old-style function declarations can also be used.)

Line 7: A check that the argument supplied is positive, as required by the rules that we are trying to enforce. If the condition fails, we use the special `iic_error` function to print an Insure++-style error message, using standard `printf` notation.

Line 10: This (apparently recursive) call to the `mymalloc` function is where the actual call to the function will be made when this interface is expanded. It appears just as in the function declaration.

Line 11: If the return value from the function call is not zero, we use the `iic_alloc` function to indicate that a block of uninitialized memory of the given size has been allocated and is pointed to by the pointer `retp`.

Line 12: The interface description ends by returning the same value returned from the call to the actual function in Line 10.

If you compile and link this interface description into your program (using the techniques described in “Using interfaces” on page 122), Insure++ will automatically check for all the requirements whenever you call the function.

```

1:      /*
2:      * File: mymal.c
3:      */
4:      #include <stdlib.h>
5:
6:      char *mymalloc(int n)
7:      {
8:          return (char *)malloc(n);
9:      }

```

## A C++ example

```

1:      /*
2:      * File: bag.h
3:      */
4:      class Bag {
5:          struct store {
6:              void *ptr;
7:              store *next;
8:          };
9:          store *top;
10:     public:
11:         Bag() : top(0) { }
12:         void insert(void *ptr);
13:     };

1:      /*

```

```

2:      * File: bag.C
3:      */
4:      #include "bag.h"
5:
6:      int main(void) {
7:          Bag bag;
8:
9:          for (int i = 0; i < 10; i++) {
10:             int *f = new int;
11:             bag.insert(f);
12:          }
13:          return 0;
14:      }

```

```

1:      /*
2:      * File: bagi.C
3:      */
4:      #include "bag.h"
5:
6:      void Bag::insert(void *ptr) {
7:          store *s = new store;
8:          s->next = top;
9:          top = s;
10:         s->ptr = ptr;
11:         return;
12:     } \

```

Let's assume that `bagi.C` is a part of a class library that was not compiled with Insure++, e.g. a third-party library. We can simulate this situation by compiling the files with the following commands:

```

insure -g -c bag.C
CC -c bagi.C
insure -g -o bag bag.o bagi.o

```

An interface for the `insert` class function might look like this:

```

1:      /*
2:      * File: bag_i.C
3:      */
4:      #include "bag.h"
5:
6:      void Bag::insert(void *ptr) {
7:          iic_save(ptr);

```

```

8:             insert(ptr);
9:             return;
10:          }

```

We can then compile the interface file with the `iic` compiler as follows:

```
iic bag_i.C
```

To get Insure++ to use the new interface description, we need to use the following compilation commands in place of the earlier commands:

```
CC -c bagi.C
insure -g -o bag bag.C bagi.o bag_i.tqs
```

## The basic principles of interfaces

As shown in the previous examples, interface descriptions have the following elements:

- The declaration of the interface description looks just like a piece of C code for the described function. It declares the arguments and return type of the function. Either ANSI or Kernighan & Ritchie style declarations may be used, but ANSI style is preferred, since K&R style declarations have implicit type promotions.
- The body of the interface description uses calls to functions whose names start with `iic_` to describe the behavior of the routine.
- The interface function appears to call itself at some point.

These concepts are common to all interface descriptions.

## Interface creation strategy

There are several possible strategies for creating interfaces for your software depending on what resources you have available and how much time you wish to spend on the project.

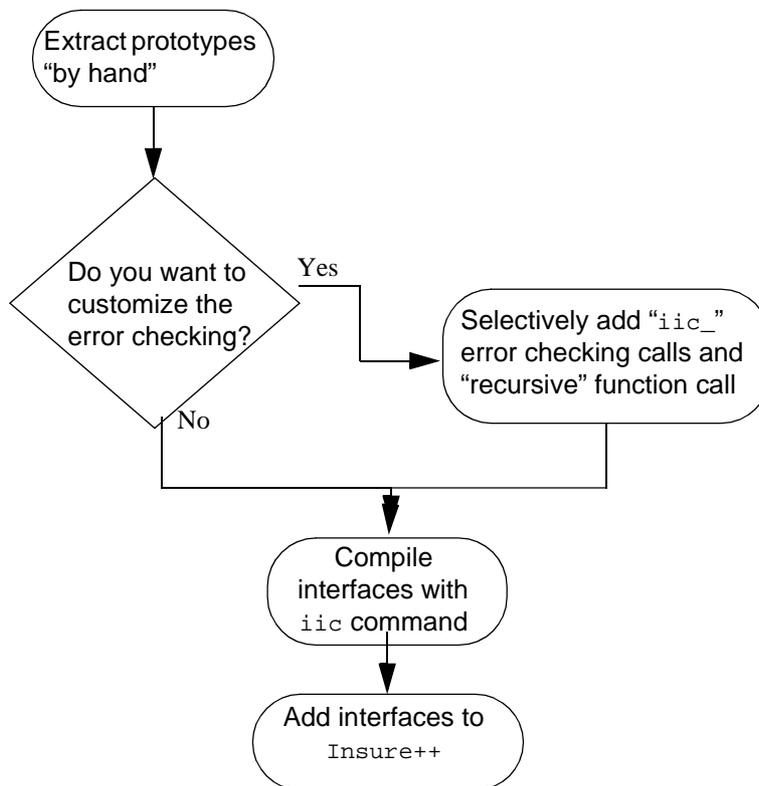
Normally, we recommend the following steps:

- Create a file containing ANSI-style prototypes for the functions for which you want to make interfaces.

- Extend these prototypes by adding additional error checks with the built-in `iic_` functions.

Getting to the first stage will allow you to perform strong type-checking on all the functions in your application. Going to the second stage provides full support for all of Insure++’s error checking capabilities.

Various aids are provided to help you implement these two stages, as briefly summarized in the flowchart in the figure below which includes page references for the most important steps.



## Trivial interfaces - function prototypes

The interfaces described so far have been “complete” in the sense that they contain error checking calls and also the “fake” recursive call typical of an interface function. There is actually one level of interface that is even simpler than this--an ANSI-style function prototype.

If you make a file containing ANSI-style prototypes for all of your functions, compile it with the `iic` program, and then add it to your `insure` command (as described on “Using interfaces” on page 122), you will get strong type-checking for all of your functions. You can then incrementally add to this file the extended interface descriptions with better memory checking and the “fake” recursive call.

## Using `iiwhich` to find an interface

The simplest way to generate an interface is to copy one from a routine that does something similar. In the two examples which started this section, we used interfaces to functions that behaved roughly the same way that `malloc` and `memcpy` operate. Furthermore, these two system functions are ones that Insure++ knows about automatically, because interfaces to all system calls are shipped with Insure++.

To see how their interfaces are defined, we use the command `iiwhich` as follows

```
iiwhich malloc memcpy
```

The output from this command is shown below.

```
1:      malloc: Interface in
          /usr/local/release/lib.solaris/cc/libc.tqi
2:
3:      void *malloc(size_t size)
4:      {
5:          void *r;
6:
7:          ;
8:          if (size < 0)
9:              iic_error(USER_ERROR, "Neg-
ative size
                                passed to malloc: %ld",
```

```

10:                                     (long) size);
11:         r = malloc(size);
12:         if (!r) {
13:             if
(!_Insight_is_direct_linked()) {
14: iic_error(RETURN_FAILURE,
                                     "malloc(%u)
failed: %s",
15:                                     size,
sys_errlist[errno]);
16:     }
17:     } else {
18:         ;
19:         iic_alloc(r, size);
20:         iic_write_uninit_pattern(r,
size);
21:     }
22:     ;
23:     return r;
24: }
25:
26: memcpy: *Linkable* interface in
/usr/local/release/lib.solaris/cc/libc.tqi
27:
28:     void * memcpy(void *d, void *s, size_t len);

```

**Note:** On your system, these may be linkable interfaces. If this is the case, you will need to find the source code to these interfaces in the `src.$ARCH/$COMPILER` directory).

The first block of “code” is the interface which defines the behavior of the `malloc` function, and the second describes `memcpy`. Note that they both follow the principles described above. They look more or less like C code with one strange exception: each function appears to call itself!

This is not recursive behavior, because this is not real C code. What really happens is that calls to the functions shown are *replaced* by the interface code. Nonetheless, it can be thought of as C code when you write your own interfaces.

A second slightly tricky feature concerns the behavior of function calls made within an interface definition. These are of two types:

- Calls to `Insure++` interface functions, whose names begin with `iic_`, are detected by the `iic` command and turned into sequences of error checking calls. They are not real function calls in themselves.
- Calls to other functions are made exactly as requested. This can be a problem if you end up passing a bad pointer to an unchecked library call, which may cause the program to fail before `Insure++` can print an error message.

Note that the `iiwhich` command is also useful if you want to see what properties of a function are being checked by `Insure++`, or if `Insure++` knows anything about it. The command

```
iiwhich foo
```

shows you the interface for the function `foo`, if it exists. If no interface exists, no checking will be done on calls to this function unless you write an interface yourself.

## Writing simple interfaces

Using `iiwhich` can save you a lot of time. Before starting to write your own interface files, particularly for system functions, you should check that one hasn't already been defined. Then, if you can think of a common function that operates in a similar way to the function you're trying to interface, start by copying its definition and modifying it. In either case, you must understand the way that the interfaces work, and to do this, you must first understand their goal.

The `malloc` function returns blocks of memory, and we need to tell `Insure++` about the size and location of such blocks. This is the reason for the call to `iic_alloc` at line 9. This is the interface function that tells `Insure++` to record the fact that a block of uninitialized memory of the given size has been allocated. From then on, references to this block of memory will be understood properly by `Insure++`.

Similarly, the purpose of `memcpy` is to take a number of bytes from one particular location and copy them to another. This activity is indicated by the call to the interface function `iic_copy` at line 32. `Insure++` uses this call to understand that two memory regions of the indicated size will be read and written, respectively.

The other code shown in the interface descriptions is used to check that parameters lie in legal ranges and is used to provide additional error checking.

## Using interfaces

To use an interface, we first compile it with the `Insure++` interface compiler, `iic`. If, for example, we put the interface for the `lib_gimme` function in a file called `gimme_i.c`, we would use the command

```
iic gimme_i.c
```

This results in the file `gimme_i.tqs`, which can be passed to `Insure++` on the command line as follows:

```
insure -c gimme_i.tqs wilduse.c
insure -o wild wilduse.o mylib.a
```

in which we assume that the library containing the actual code for the `lib_gimme` routine is called `mylib.a`.

An additional example of how to use an interface can be found earlier in this section “A C++ example” on page 115.

The basics for using an interface, therefore, are to:

- Compile the interface with `iic`.
- Recompile your program.

Note that you don’t have to limit yourself to a single interface per source file. If you are preparing an interface module for an entire library, or a source file with multiple functions, you can put them all into the same interface description file.

Similarly, you don’t have to pass all the names of your compiled interface modules on the `insure` command line every time. You can add lines to your `.psrc` files that list interface modules as follows

```
insure++.interface_library /usr/trf/mylib.tqs
insure++.interface_library /usr/local/ourlib.tqs
```

## Ordering of interfaces

Files containing compiled interface definitions can be placed in any directory. Insure++ can be told to use such files in various ways, and processes them according to the following rules:

- If a standard library interface exists it is processed first.
- Interfaces specified in `interface_library` statements in configuration (`.psrc`) files are processed next, potentially overriding standard library definitions.
- Interface modules (i.e., files with the suffix `.tqs` or `.tqi`) specified on the `insure` command line override any other definitions.

Later definitions supercede earlier ones, so you can make a local definition of a library function and it will override the standard one in the library.

To see which interface files will be processed, and in which order, you can execute the command

```
iiwhich -l
```

which lists all the standard library files for your system, and then any indicated by `interface_library` commands in configuration files.

To find a function in an interface library, you can use the `iiwhich` command as already described. To list the contents of a particular `.tqs`, `.tqi`, or `.tql` file, use the `iiinfo` command.

## Working on multiple platforms or with multiple compilers

Many projects involve porting applications to several different platforms or the use of more than one compiler. Insure++ deals with this by using two built-in variables, which denote the machine architecture on which you are running and the name of the compiler you are using. You can use these values to switch between various combinations, each specific to a particular machine or compiler.

For example, environment variables, `~s` (for `HOME` directories) and the `%` notation described on “Options used by Insure++” on page 179, are expanded when processing filenames, so the command

```
insure++.interface_library $HOME/insure++/%a/%c/foo.tqs
```

loads an interface file with a name such as

```
/usr/me/insure++/sun4/cc/fo0.tqs
```

in which the environment variable `HOME` has been replaced by its value and the ‘`%a`’ and ‘`%c`’ macros have been expanded to indicate the architecture and compiler name in use. This allows you to load the appropriate `.tqs`, `.tqi`, or `.tql` files for the architecture and compiler that you are using.

**Note:** One problem to watch out for occurs when you switch to a compiler for which Insure++ supplies no interface modules. In this case, you will see an error message during compilation. Several work-arounds are possible as described in the FAQ (`FAQ.txt`).

## Common interface functions

Most definitions need only a handful of interface functions of which we’ve already introduced the most common:

```
void iic_alloc(void *ptr, unsigned long size);
```

Declares a block of dynamically allocated, uninitialized, memory.

```
void iic_source(void *ptr, unsigned long size);
```

Declares that a block of memory is read.

```
void iic_sourcei(void *ptr, unsigned long size);
```

Declares that a block of memory is read and also checks that it is completely initialized.

```
void iic_dest(void *ptr, unsigned long size);
```

Declares that a block of memory is modified.

```
void iic_copy(void *to, void *from,
unsigned long size);
```

Declares that the indicated block of memory is copied.

```
void iic_error(int code, char *format, ...);
```

Causes an error to be generated with the indicated error code.

Subsequent arguments are treated as though they were part of the `printf` statement.

Other commonly occurring functions are listed below together with examples of system calls that use them. You can use the `iiwhich` command on the listed functions to see examples of their use.

```
int iic_string(char *ptr, unsigned long size);
```

Declares that the argument should be a `NULL` terminated character string. This is used in most of the string handling routines such as `strcpy`, `strcat`, etc. The second argument is optional, and can be used to limit the check to at most `size` characters.

```
void iic_alloci(void *ptr, unsigned long size);
```

Declares a block of dynamically allocated, initialized memory such as might be returned by `calloc`.

```
void iic_allocs(void *ptr, unsigned long size);
```

Declares a pointer to a block of statically allocated memory. Used by functions that return pointers into static strings. `ctime` and `getenv` are examples of system calls that do this.

```
void iic_unalloc(void *ptr);
```

Declares that the indicated pointer is de-allocated with the system call `free`.

A complete list of available functions is given in “Interface Functions” on page 353.

## Checking for errors in system calls

We can make interfaces even more user-friendly by adding checks for common problems, similar to the user level checks that were described in “Code Insertions” on page 109.

For example, `malloc` can fail. This is the reason for the second branch of the code in line 11. If the actual call to `malloc` fails, instead of telling Insure++ about a block of allocated memory with `iic_alloc`, we cause an Insure++ error with code `RETURN_FAILURE` and the error message shown. This, in turn, will cause a message to be printed (at runtime) whenever `malloc` fails and the `RETURN_FAILURE` error code has been unsuppressed. (See “Enabling error messages” on page 76.)

Similarly, `memcpy` can cause undefined behavior when given perfectly valid buffers that happen to overlap. We check for this case in the code at line 20, and again, cause an `Insure++` error if a problem is detected.

This method provides a very powerful debugging technique, which is used extensively in the interface files supplied with `Insure++`. Since the `RETURN_FAILURE` error code is suppressed by default, you will normally not be bothered by messages when system calls fail. The assumption is that the user application is going to deal with the problem. In fact, it may require certain system calls to fail in order to work properly. However, when particularly nasty bugs appear, it is often useful to enable the `RETURN_FAILURE` error category to look for cases where system calls fail “unexpectedly” and are not being handled correctly by the application. Errors such as missing files (causing `fopen` to fail) or insufficient memory (`malloc` fails) can then be diagnosed trivially.

## Using `Insure++` in production code

A particularly powerful application of the technique described in the previous section is to make two different versions of your application:

- One with full error checking
- One without `Insure++` at all

The first of these is used during application development to find the most serious bugs. The second is the one that will be used in production and shipped to customers.

When you or your customer support team is faced with a problem, they can run this code with the `RETURN_FAILURE` error class enabled and look for “unexpected” failures such as missing files, incorrectly set permissions, insufficient memory, etc.

## Advanced interfaces: complex data types

The interfaces that have been considered so far are simple in the sense that their behavior is determined by their arguments in a straightforward manner.

To show a more complex example, consider the following data structure

```
struct mybuf {
    int len;
    char *data;
};
```

This data type could be used to handle variable length buffers. The first element shows the buffer length and the second points to a dynamically allocated buffer.

The code which allocates such an object might look as follows:

```
#include <stdlib.h>

struct mybuf *mybuf_creat(n)
    int n;
{
    struct mybuf *b;

    b = (struct mybuf *)malloc(sizeof(*b));
    if(b) {
        b->data = (char *)malloc(n);
        if(b->data) b->len = n;
        else b->len = 0;
    }
    return b;
}
```

Similarly, we might define operations on a `struct mybuf` that work in quite complex ways on its data.

To build an interface description of the `mybuf_create` function which detailed all its behavior would require the following code

```
struct mybuf *mybuf_creat(n)
    int n;
{
    struct mybuf *b;

    b = mybuf_creat(n);
    if(b) {
        iic_alloci(b, sizeof(*b));
        if(b->data)
            iic_alloc(b->data, b->len);
    }
    return b;
}
```

Note how the structure of the interface description follows that of the original source.

This matching would be seen in the interface descriptions of all the other functions that operate on the `struct mybuf` data type, too. In fact, the interface description would probably end up looking quite a lot like the source code!

There are three standard approaches to dealing with this problem:

- Forget the interface entirely, process the real source code with `insure`, and link it in the normal manner.
- “Go deep” and define an interface that mimics all of the details of the interface, including all the operations on the internal structure elements.
- “Go opaque” and build an interface that defines some levels of the functions without necessarily going into details of their action.

Each of these is a good approach in a different situation.

The first approach is the best in terms of accuracy and reliability. Given the original source code, `Insure++` will have complete knowledge of the workings of the code and will be able to check every detail itself.

The second approach is best when the source code is unavailable but you still want to check every detail of your program’s interaction with the affected routines. It can be implemented only if you have intimate knowledge of how the routines work, since you will have to use the interface functions to mimic the actions of the functions on the individual elements of the `struct mybuf`.

The third approach is appropriate when you are sure that the functions themselves work correctly. Perhaps, for example, you’ve been running `Insure++` on their source code at some earlier date and you know that they are internally consistent and robust. In this case, you may want to increase the performance of the rest of your program by checking the high level interface to the routines, but not their internal details.

Another reason for adopting the third approach might be that you actually don’t know the details of the functions involved and might not be able to duplicate their exact behavior. A good example would be building an interface to a third party library. You have clear definitions of the upper level behavior of the routines, but may not know how they work internally.

The first and second approaches have already been discussed. The third approach is easily achieved by doing nothing - Insure++ will recognize that the data type has not been declared in detail and should therefore not be checked in detail. You can choose for yourself which fields to declare in detail and which to ignore.

## Additonal Information

Since it is possible to express a wide range of actions in C, interface files must have correspondingly sophisticated capabilities in order to define their actions and check their validity.

One of these features was seen in the previous section: the `iic_startup` function. This function can be defined in any interface file and contains calls to interface functions that will be made before calling *any* of the other functions defined in the interface file. Typically, you will place definitions and initializations of known global or external variables in this function.

Note that each interface file may have its own `iic_startup`.

Variable argument lists are dealt with by using the pre-defined variable `__dots__`. For example, the interface specification for the standard system call `printf` is

```
int printf(char *format, ...)
{
    iic_string(format);
    iic_output_format(format);
    return(printf(format, __dots__));
}
```

The variable `__dots__` in the function call matches the variable arguments declared with “...” in the definition.

Checking of `printf` and `scanf` style format strings is done with the `iic_output_format` and `iic_input_format` routines. These check that arguments match their corresponding format characters. `iic_strlenf` returns the length of a string after its format characters have been expanded and can be used to check that buffers are large enough to hold formatted strings.

A complete list of interface functions can be found in “Interface Functions” on page 353.

## Callbacks

In many programming styles, such as programming in the X Window System or when using signal handlers, functions are registered and are then “called-back” by the system. Often the user program contains no explicit calls to these functions.

If the callback functions use *only* variables that are defined in the user program, nothing unusual will happen, since Insure++ will understand where all this data came from and will keep track of it properly. In many cases, however, the library function making the callback will pass additional data to the called function that was allocated internally, which Insure++ never saw.

For example:

- UNIX functions such as `qsort` and `scandir` take function pointer arguments that are called back from within the system function.
- Signal handling functions often pass to their handlers a data structure containing hardware registers and status information.
- The X Window System library often passes information about the display, screen, and/or event type to its callback functions.

In these cases, Insure++ will attempt to look up information about these data structures without finding any, which limits its ability to perform strong error checking.

**Note:** This is not a serious limitation - it merely means that the unknown variables will not be checked as thoroughly as those whose allocation was processed by Insure++.

If you wish to improve the checking performed by Insure++ in these cases, you can use the interface technology in two different ways:

- You can make interfaces to the functions that install or register the callbacks (with `iic_callback`) indicating how to process their arguments when the callbacks are invoked.
- You can make interface definitions for your callback functions themselves, adding the keyword `iic_body` to their definition.

These two options are described in the next sections.

## Using iic\_callback

The first of these approaches is more general, since it allows you to define, in a single interface specification, the behavior of *any* callback that is installed by the function specified. To see how this works, consider the standard utility sorting function, `qsort`. One of the arguments to this routine is a function pointer that is used to compare pairs of elements during sorting.

The following interface to this function checks that the `qsort` function does no more than  $N^2$  comparisons, where  $N$  is the number of elements (this may or may not be a sensible check, but serves the purpose of explaining callback interfaces):

```

1:      #include <sys/stdtypes.h>
2:      #include <math.h>
3:
4:      static int _qsort_num_comparisons;
5:
6:      static int _qsort_cb(void *e1, void *e2)
7:      {
8:          _qsort_num_comparisons += 1;
9:          return _qsort_cb(e1, e2);
10:     }
11:
12:     void qsort(void *base, size_t nelem,
13:               size_t width,
14:               int (*func)(void *, void *))
15:     {
16:         iic_dest(base, nelem*width);
17:         iic_func(func);
18:         iic_callback(func, _qsort_cb);
19:         _qsort_num_comparisons = 0;
20:         qsort(base, nelem, width, func);
21:         if (_qsort_num_comparisons >
22:             nelem
23:             * nelem)
24:             iic_error(USER_ERROR,
25:                      "Qsort took %d compares.",
26:                      _qsort_num_comparisons);
27:     }

```

The main body of the interface is in lines 16-25.

Line 16 checks that the pointer supplied by the user indicates a large enough region to hold all the data to be sorted, while line 17 checks that the function pointer actually points to a valid function. Line 20 contains the call to the normal `qsort` function.

The interesting part of the interface is the call to `iic_callback` in line 18. The two arguments connect a function pointer and a “template”, which in interface terms is the name of a previously declared static function; in this case `_qsort_cb`, declared in lines 6-10. The template tells Insure++ what to do whenever the system invokes the called-back, user-supplied function. In this particular case, the interface merely increments a counter so we can see how many times the callback gets called (note that we set the counter to 0 on line 19 of the `qsort` interface). In general, you can make any other interesting checks here before or after invoking the callback function.

Notice that once this interface is in use, it automatically processes *any* function that gets passed to the `qsort` function.

## Using `iic_body`

The second callback option is to define interfaces for each individual function that will be used as a callback.

Consider, for example, the X Window System function `XtAddCallback`, which specifies a function to be called in response to a particular user interaction with a user interface object. It is quite common for code to contain many calls to this function, for example

```
XtAddCallback(widget, ..., myfunc1, ...);
XtAddCallback(widget, ..., myfunc2, ...);
XtAddCallback(widget, ..., myfunc3, ...).
```

One solution for this routine would be to provide an `iic_callback` style interface for the `XtAddCallback` function as described in the previous section. The second method is to specify interfaces to the called-back functions themselves, with the additional `iic_body` keyword. An interface for the routine `myfunc1` might be written as follows:

```
/*
 * Interface definition for callback function
 * uses the iic_body keyword.
 */
```

```

void iic_body myfunc1(Widget w,
                    XtPointer client_data,
                    XtPointer call_data)
{
    if (!call_data)
        iic_error(USER_ERROR,
                  "myfunc1 passed NULL call_data");
    myfunc1(w, client_data, call_data);
    return;
}

```

This interface checks that `myfunc1` is never passed `NULL` `client_data`.

Note that in this scenario you would have to specify three separate interfaces; one each for `myfunc1`, `myfunc2` and `myfunc3`. (And, indeed, any other functions used as callbacks.)

## Which to use: `iic_callback` or `iic_body`?

From the previous discussion it might seem that `iic_callback` should always be preferred over `iic_body`, since it is more general and less code must be written. Unfortunately, the general `iic_callback` method has a severe limitation: the code generated by Insure++ when you use `iic_callback` is good for “immediate use only”.

To understand what this means, consider the difference between the two cases already discussed:

- In the `qsort` example, the `iic_callback` function made the association between function pointer and template, which was then immediately used by the `qsort` function. By the time the interface code returns to its caller, the connection between function and template is no longer required.
- In the X Window System example, the callbacks registered by the `XtAddCallback` function are expected to survive for the remainder of the application (or until cancelled by another X Window System call). Similarly, the connection between function pointer and template is expected to survive as long.

As a consequence, the `iic_callback` method is only applicable to a small number of circumstances, and in general you must either:

- Use the `iic_body` method
- Do nothing, and allow Insure++ to skip checks on unknown arguments to callback functions.

## Conclusion

Interfaces play an important, but optional, role in the workings of Insure++.

If you wish, you can always eliminate error messages about library calls by adding `suppress` options to your `.psrc` files and running your program again. This approach has the advantage of being very quick and easy to implement, but discards a lot of information about your program that could potentially help you find errors.

To capture all the problems in your program, you need to use interfaces. Insure++ is supplied with interfaces for all the common functions and quite a few uncommon ones. These are provided in source code form in the directory `src.$ARCH/$COMPILER` so that you can look at them and modify them for your particular needs.

The `iiwhich` command can help you find existing definitions that can then be used as building blocks in making your own interfaces.

If you build an interface to a library that you'd like to share with other users of Insure++, please send it to us ([insure@parasoft.com](mailto:insure@parasoft.com)) and we'll make it available.

# Introducing Inuse

Inuse is a graphical tool designed to help developers avoid memory problems by displaying and animating in real time the memory allocations performed by an application.

By watching your program allocate and free dynamic memory blocks, you gain a better understanding of the memory usage patterns of your algorithms and also an idea of how to optimize their behavior.

Inuse allows you to

- Look for memory leaks.
- See how much memory your application uses in response to particular user events.
- Check on the overall memory usage of your application to see if it matches your expectations.
- Look for memory fragmentation to see if different allocation strategies might improve performance.

## Running the Inuse user interface

The Inuse graphical user interface (GUI) does nothing but wait for programs to start and connect to it, at which point it can display their memory activity. When these programs terminate, the Inuse window remains active until you choose to exit it. This allows you to analyze the data gathered during a program's run once the program has completed execution.

If you exit Inuse while a program is still running, that program will continue running as usual but will stop sending memory activity data. You will have to start the inuse process again before memory activity can be displayed.

To run the example shown in this section, execute the commands

```
cp /usr/local/inuse/examples/c/slowleak* .inuse
```

The first of these commands copies a set of example files to your local working directory, while the second starts Inuse. Normally, you will only

have to execute the `inuse` command once. The `inuse` process will remain running in the background, accepting display requests from any application that you choose to run.

## Compiling and linking for Inuse

To use Inuse, you need to link your executable program with the `insure++` command and the `-Zuse` option. Compile the objects and libraries that make up your application with your regular compiler and link them with `Insure++` to create a new executable.

**Note:** The `insure++` command replaces the `insight` command used in previous versions of Inuse.

Compile and link the sample program with the `Insure++` command:

```
cc -c slowleak.c
insure++ -Zuse -o slowleak slowleak.o
```

**Note:** If you are using a compiler other than `cc`, you can tell `Insure++` to use the correct compiler during the link step by adding a line such as `insure++.compiler gcc` to a `.psrc` file.

The first command compiles the source file into an object module, while the second links with the special Inuse dynamic memory library.

Inuse is already available to your program if you are compiling your program with `Insure++` to debug your code. With earlier versions, if you compiled with `Insure++`, Inuse would also display information about the `Insure++` runtime as it performs error detection. This is no longer the case, because `Insure++` now uses two separate heaps for the program. Nevertheless, we still recommend the method described above.

## Enabling runtime activity display

Now that your program is linked with the appropriate libraries, you will need to enable the runtime memory activity display. To do this, you must add the following option to your `.psrc` file.

```
insure.inuse on
```

## Running the application

Once you have enabled the runtime display, you can run your application just as you would normally. To run the example application `slowleak`, type the command

```
slowleak
```

## The Inuse display

When you start the example application, it will connect to Inuse. The Inuse display shows which applications are currently linked to the GUI. From this screen you can open any of Inuse's visual reports.

For a complete description of the Inuse display, see "Running Inuse" on page 140.

**Note:** If no connection appears, consult the "Inuse" section of the FAQ (Frequently Asked Questions) shipped with your distribution (`FAQ.txt`) or the up-to-date copies kept on our ftp and web servers.

## Is there a bug in the slowleak program?

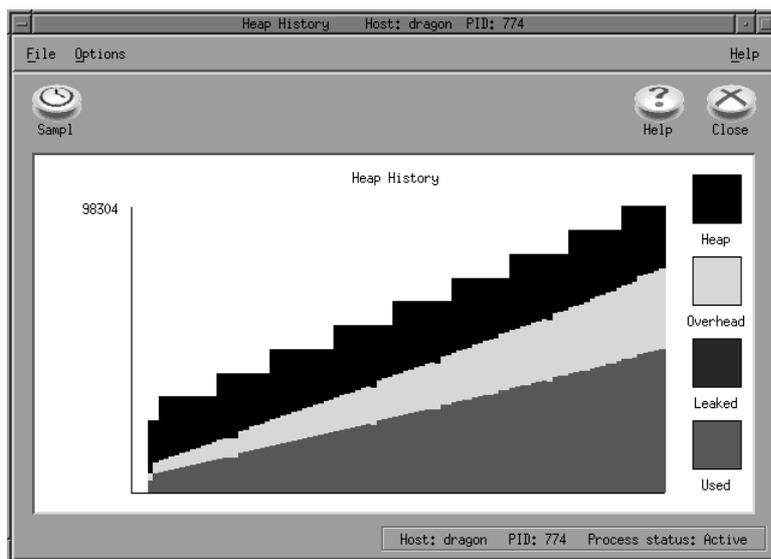
Clicking on the **Hist** button on the Inuse display will open up the **Heap History** window.

Watch the window for a few moments as the `slowleak` program continues to run. The window should soon appear similar to the one shown below in Figure 1.

Figure 1 clearly shows that the program is continuously allocating more and more memory - the classic symptom of a memory leak. This type of pattern in an Inuse report is important to watch for in your own applications, since it probably means that something is wrong.

To find the cause of the problem, you can either look at the source code manually and attempt to figure it out yourself, or simply compile the program and run it with Inuse++ using the following commands

```
insure -g -o slowleak slowleak.c
slowleak
```



**Note:** To use Inuse, you only need to link with the `insure++` command. To find the memory leak, you need to compile and link with Inuse++, as shown above.

(For a description of detecting memory leaks with Inuse++, see “Memory leaks” on page 39.)

To see the difference in Inuse’s output for correct and incorrect programs, you can either fix the problem in the `slowleak` example or copy, compile, and link the corrected version, `noleak`, with the following commands:

```
cp /usr/local/insure/examples/c/noleak.c .
cc -c noleak.c
insure -Zuse -o noleak noleak.o
noleak
```

**Note:** If you exited Inuse, you will need to start it again before running the last of the commands shown above (see “Running the Inuse user interface” on page 135). You also need to have the `insure.inuse` option set in your `.psrc` file to enable the graphical display, as explained in “Enabling runtime activity display” on page 136.



# Running Inuse

The basic steps involved in using Inuse are:

- Running the GUI.
- Linking your program to Inuse.
- Adding the `inuse on` option to your `.psrc` file and running the application program. During runtime, you can view and manipulate the displays shown by the GUI. You can even watch the memory allocation as you single step through your program from a standard code-oriented debugger.

This section will cover each of these steps in detail.

## The basics

You must start the Inuse program before you attempt to display results from any user application. (If you try to run an application before starting Inuse it will run normally, without displaying any memory activity.)

In normal use, you should enter the `inuse` command once and simply leave it running as a background process.

```
inuse
```

You should compile your code with your regular compiler and then link with Insure++ as shown below.

```
insure -o foo foo.c
```

To enable runtime display of memory activity, you need to set the following option in your `.psrc` file.

```
insure++.inuse on
```

Inuse can be linked simultaneously with any number of application programs. By turning this option on and off, you can control when your programs connect to Inuse. If you exit Inuse, you must restart it before running any applications for which you wish to display memory activity.

## The Inuse GUI

Executing the `inuse` command opens the Inuse GUI.

When you connect a program to Inuse, the connection will appear in the main window. The “plugged in” symbol next to the connection shows that the program is currently sending data to Inuse. If you tell Inuse to stop receiving memory data from the program, this symbol will change to a stop sign. When the program finishes its run or is terminated, the symbol is replaced with a “RIP”.



**Note:** The “look and feel” of a windowing application will vary quite significantly from system to system. As a result, the version of the window that you see might differ from that shown above.

### Menus

**File** - Select **Quit** to close Inuse.

**Processes** - Select:

- **Next** or **Prev** to choose among linked programs.
- **Del** to remove a program from the list.
- **Stop** to stop receiving memory display information from a linked program.

- **Step** to display one allocation or free request from the program at a time.

**Reports** - Select a report for Inuse to display. Reports are described in detail in the next section.

**Help** - Get help for using Inuse, including the **Inuse FAQ** and version information.

### Tool bar

The tool bar has buttons for most menu options. Clicking a button selects that option.

## Inuse reports

Inuse can generate the following reports:

### Heap History

This graph displays the amount of memory allocated to the heap and the user process as a function of real (i.e., 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.

You can scan through different areas of the layout by pressing the Fast Left (**F.left**), Fast Right (**F.right**) and **left** and **right** buttons on the Heap Layout tool bar. You can also zoom in (+) and out (-) of the layout by pressing the **zoom** buttons. (These options are also available in the **Controls** menu.)

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  $(total-in\_use)/total$ ).

## 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.

## Which report to use?

Which report is most useful depends on what you are trying to learn about your application.

- If you simply wish to see how much memory is being used, try the usage summary and heap history reports.
- If you wish to optimize your memory allocation strategy, perhaps by building your own allocator for blocks of certain sizes, the block frequency graph is appropriate.
- To study the heap fragmentation caused by your algorithm, and understand the way that memory blocks are laid out, you should use the heap layout graph.
- To see correlations between block id, block size, and stack traces in your program, use the query option.

## Block color in Inuse

An important visual aid in Inuse is the use of colors to represent the various properties of the heap. These colors are as follows:

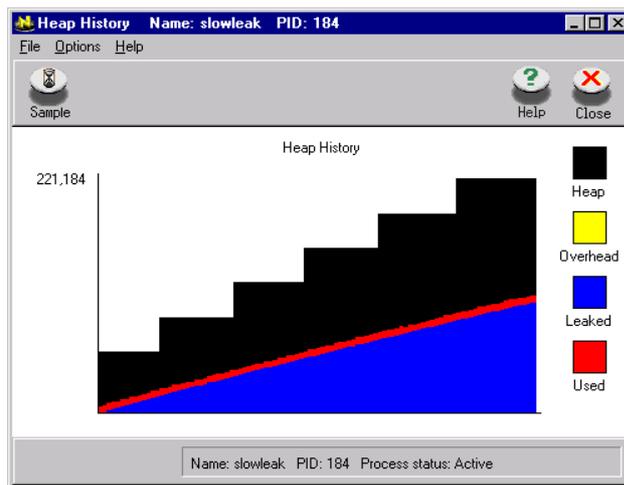
Black	Indicates the total memory allocated to the heap. This is usually the amount of memory that gets swapped to disk whenever your application is swapped from memory, regardless of whether or not you are actually using it.
Blue	Denotes leaked blocks as reported by Insure++. (Only available if running Inuse and Insure++ together).
Green	Free space that is available to be allocated.
Red	Denotes allocated blocks.
Yellow	"Overhead" associated with each block. Normally the system keeps a small amount of memory with each allocated (and maybe free) block for its bookkeeping information. This memory cannot be used by the application, although its impact can be reduced by allocating fewer large blocks rather than many small ones.

# Inuse Reports

This section contains detailed descriptions of each Inuse report option.

## The “Heap History” report

Clicking the **History** button or selecting **Heap History** from the Reports menu opens the Heap History window. If your program is running when you open this report, you will be able to monitor the amount of memory allocated by the program as it executes. If the program has ended its run, you can see how much memory was allocated across the history of the run.

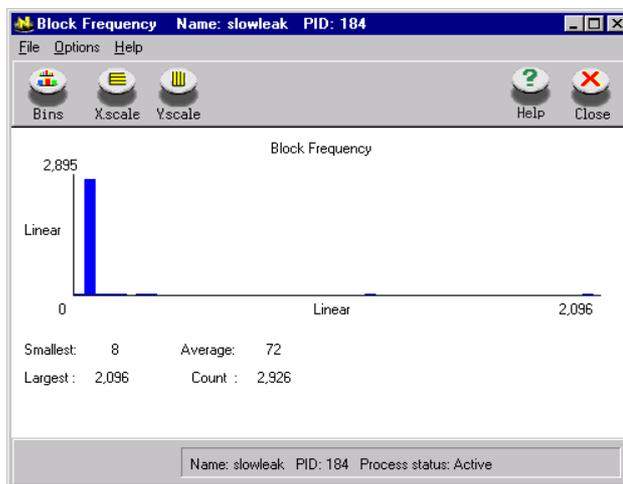


The black area indicates the total size of the heap. The red, yellow, and blue areas constitute the make-up of the heap. The red area represents the amount of memory allocated by the program. The yellow area represents the amount of “overhead” associated with the memory (but which cannot be used by the application). The blue area represents the amount of leaked memory.

To change the sampling rate, that is, how much time passes before the graph is updated, press the **Sample** button or select **Sampling** from the Options menu.

## The “Block Frequency” report

The “block frequency” report will show you what sizes of blocks are typically being allocated by your program. If you are allocating many small blocks, you may want to switch to a different memory allocation scheme which groups many small allocations into several larger ones. A “block frequency” graph might look like the one below.



The Block Frequency graph groups together blocks that are similarly sized. Each bin (column) in the graph represents the number of blocks in a particular size range. Clicking a bin will show you the number and size range of blocks contained in that bin.

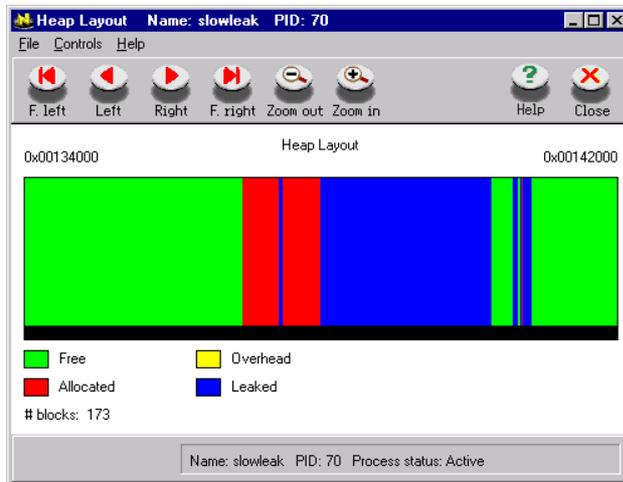
The right-most bin includes all blocks above a certain size. If this bin is very high, click to see the size range covered by that bin. If the range is fairly wide, you can rescale the graph to include more bins by pressing the **Bins** button or selecting **Bins** from the Options menu. Entering a number *higher* than the number currently shown should *narrow* the size

range for each bin. Likewise, entering a number that is *lower* than the number currently shown should *increase* the size range included in each bin. If all block sizes are currently represented on the graph, increasing the bin number will have no effect.

You can alternate between a linear or logarithmic scale by pressing the **Xscale** and **Yscale** buttons. Pressing the **Xscale** button will toggle the x-axis between linear and logarithmic scales. Pressing the **Yscale** button will toggle the y-axis between linear and logarithmic scales. Selecting **Horiz.log/log** or **Vert.log/log** from the Options menu will also toggle the x- or y-axis.

## The “Heap Layout” report

The Heap Layout report shows the status of blocks in the program’s heap. Blocks are either free, allocated, overhead, or leaked.



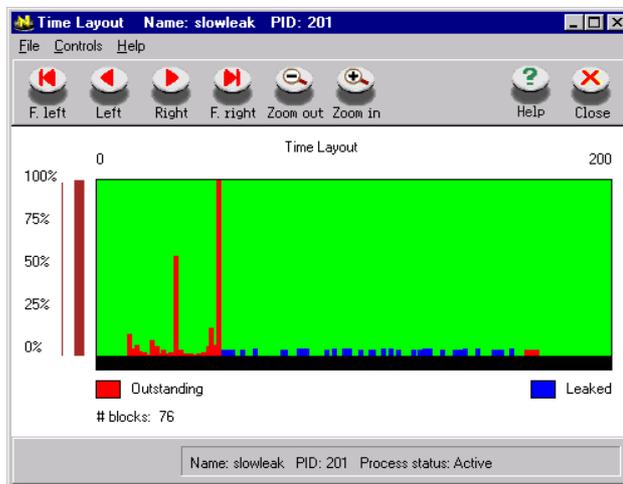
Click a block to see its address, size, and status. This information is shown in the lower right corner of the display screen.

If the block is allocated or leaked, a Memory Information window will also appear. The memory information window contains the block's id, address, stack depth, and stack trace.

You can scan through different areas of the heap layout by pressing the Fast Left (**F.left**), Fast Right (**F.right**) and **left** and **right** buttons on the Heap Layout tool bar. You can also zoom in (+) and out (-) of the layout by pressing the **zoom** buttons. (These options are also available in the **Controls** menu.)

## The “Time Layout” report

The Time Layout report shows how memory blocks are allocated across the run of the program. As each block is allocated, it is added to the end of the display. As blocks are freed, they are marked green.

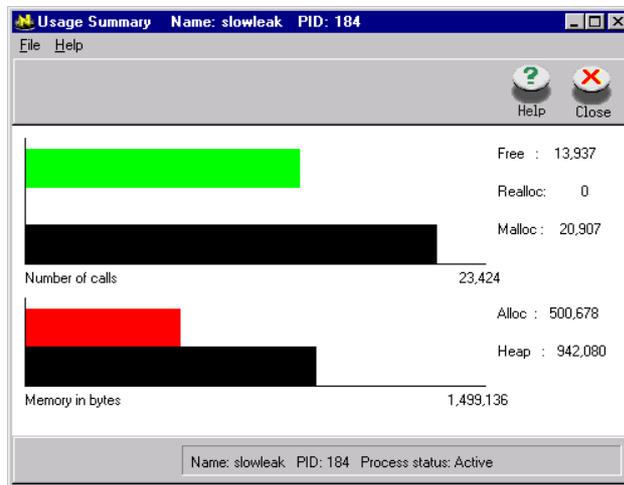


From this display, you can watch how memory is allocated over the run of your program. As you see the patterns in which memory blocks are allocated and freed over time, you can better optimize your program's use of memory. Memory leaks are also shown on this display.

You can scan through the run by pressing the Fast Left (**F.left**), Fast Right (**F.right**) and **left** and **right** buttons on the Time Layout tool bar. You can also zoom in (+) and out (-) of the layout by pressing the **zoom** buttons. (These options are also available in the **Controls** menu.)

## The “Usage Summary” report

The “usage summary” report will show you how much memory you are using and how often calls have been made to each category of memory allocation functions `malloc`, `realloc`, and `free`. Note that the `malloc` category includes all functions which allocate dynamic memory, for example `calloc`, `memalign`, and `XtMalloc`. Similarly, the `realloc` category includes all functions which relocate or resize dynamic memory blocks. The `free` category includes all functions which free dynamic memory. You can calculate the number of blocks currently allocated by subtracting the number of `free`s from the number of `malloc`'s. A typical “usage summary” graph might look like the following one below.



The number given by “Alloc” is the total number of bytes allocated dynamically by your program. The number given by “Heap” is the total

number of bytes currently allocated by the system to your program's heap.

The number given with "Number of calls" and "Memory in bytes" is simply the extreme value on the x-axis for each graph. The limit of each graph will change as Inuse updates the display with more memory allocation function calls.

## Query Reports

Query is a powerful tool that makes it easier for you to understand how memory is being used by your program. With Query, you can find out exactly how much memory is being allocated to blocks of a particular size or location; how much memory is being allocated from a particular path; find out the stack traces of blocks of a particular size or location; and much, much more.

By creating queries, you will be creating a "model" of your program's memory use. You will be able to "look" at it from different angles and approaches, learning more and more with each successive report. As you come to understand how and where your program uses memory, you will be able to better optimize your program's memory use.

For example, if the Block Frequency report shows that your program is using many small allocations (creating a lot of memory overhead), you might want to know exactly how much memory is being allocated to these small blocks. You will also want to know which part of the code is responsible for creating them.

To find out how your program is distributing memory across block sizes, you can run a query that shows how much memory is being allocated to each size block.

If you find that your small blocks contain a lot of memory, you can then run a query that gives you the block ids and stack traces responsible for creating these small blocks.

Armed with this information, you will be able to make simple adjustments to your code that will result in a more effective use of memory.

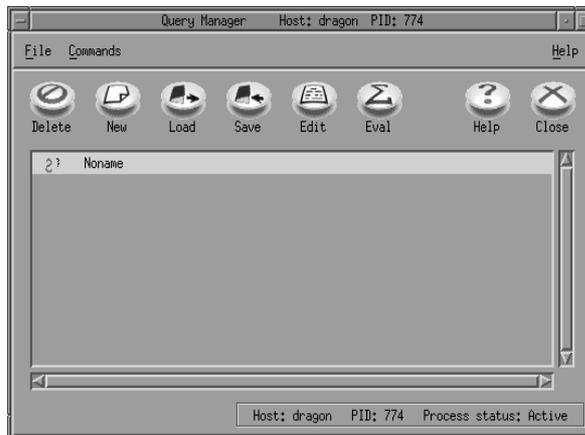
You can run queries on any combination of block id, block size, and stack trace. These queries can be as flexible or as restrictive as you choose, as you set the parameters (see “Editing a Query” on page 152).

Running queries does not affect your program or its operation. By running different queries and trying different approaches, you will soon see how valuable and informative these reports can be.

## Running a Query

Clicking **Query** opens the Query Manager screen. From here you can:

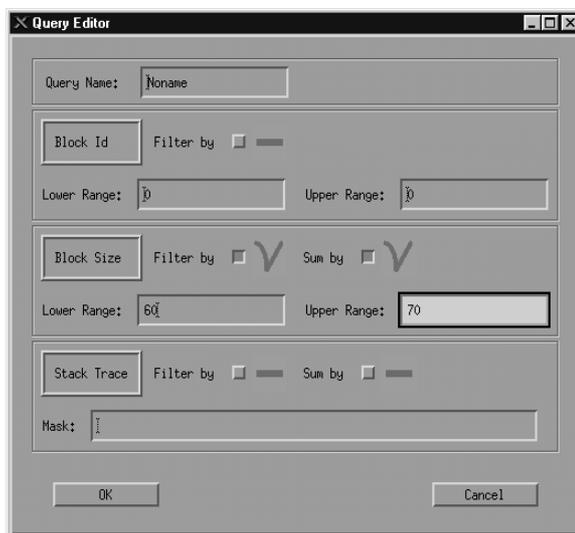
- Press **New** to start a new query.
- Press **Load** to open a saved query.
- Press **Delete** to delete the current query.
- Press **Save** to save the current query for later use.
- Press **Eval** to evaluate the current query.



When you open a query, its name will appear in the Query Manager window. If you pressed **New**, the query will be named `NoName`.

## Editing a Query

Pressing the **Edit** button on the Query Manager screen will let you edit the currently selected query.



The Query Editor window lets you:

- Change the name of a query.
- Set the lower and upper ranges for the block ids you want to isolate.
- Set the lower and upper ranges for the block sizes you want to isolate.
- Enter an expression to isolate certain stack traces.
- Choose whether to receive a complete report (including block id, block size, and stack trace data) or just a summary of block size and/or stack trace information.

The default values for a query are 0 for the lower and upper block id and block size ranges and no expressions in the stack trace area. This will

produce the widest query, listing all memory blocks allocated by your program.

You can filter a query to isolate particular block ids, block sizes, and/or stack traces. Just enter the values you want to filter for and “check” the **Filter by** box.

The **Sum by** option provides a useful summary of block size and stack trace data. Use it in combination with the **Filter by** option or on its own to get a breakdown of blocks by size and/or stack trace.

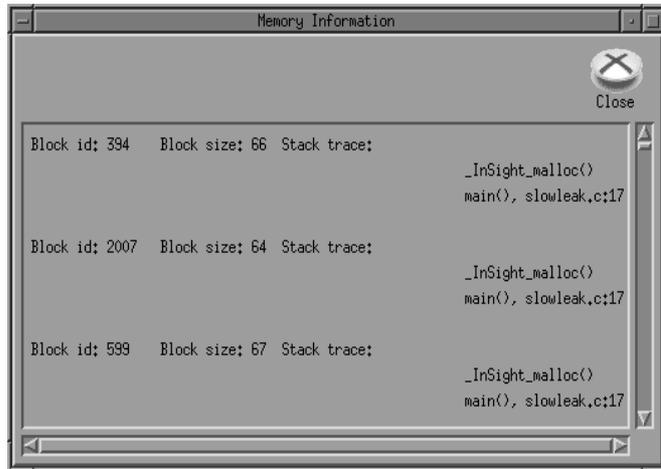
For example, let’s say you want to learn more about blocks that are between 60 and 70 bytes in size. In the Block Size area, enter 60 in the lower range and 70 in the upper range. Then click the Block Size **Filter by** box and press **OK**. Running this query will return block ids and stack traces for all blocks allocated by your program that are sized between 60 and 70 bytes.

Checking the Block Size **Sum by** box for this query will return a list showing how much memory (in bytes) is being allocated to each block size in that range.

You can enter filters for any or all three areas in a single query. This can help you find out the exact stack trace(s) responsible for the largest block allocations by your program, or conversely, to find out the size and location of memory allocated from a particular stack trace path.

Once you have edited your query, pressing **OK** returns you to the Query Manager.

To run the query, press the **Eval** button or select **Evaluate Query** from the Commands menu. Query results will appear in a text window.



# Introducing TCA

TCA (Total Coverage Analysis) lets you get “beneath the hood” of your program to see which parts of your program 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 guarantee faster delivery of more reliable programs.

## Coverage Analysis

The idea behind test coverage is to analyze how many of an application's files, functions, and statements have been executed. This data can be used during development, and particularly during testing, to give some idea of the overall quality of the testing.

The hope is that having information about which parts of an application haven't been tested will enable you to modify or enhance your existing testing procedures to cover the untested portions.

Unfortunately, this is a vain hope with conventional testing methods, because even if the effort is made to achieve 100% test coverage, the tests will usually be heavily biased towards the “important” parts of the code which get executed most often.

Because of this, many organizations do not use test coverage analysis to any great extent in their development process.

## The significance of runtime testing

An important difference emerges, however, when one considers the additional power of runtime debugging tools such as Insure++.

The quality of the checking performed by Insure++ is independent of the amount of effort put in by the development or quality assurance team--it performs complete and thorough checking on any piece of code that it executes (and also, of course, a significant amount of compile-time checking on code that it merely compiles). As a result, it makes a lot of

sense to aim for 100% test coverage if you are using a product such as Insure++, because the testing is much more thorough.

For this reason, Insure++ contains the Total Coverage Analysis (TCA) module, which is a component of the Total Quality Software package designed to guide you to 100% execution of your application during the testing and quality assurance processes.

# Using TCA

## Preparing your code for coverage analysis

The coverage analysis system works in a similar way to Insure++. Your application is processed with the `insure` command instead of your normal compiler. This process first builds a temporary file containing your original source code modified to include checks for coverage analysis, and then passes it to your normal compiler. While your application runs, these modifications cause your application to create a database containing information about which blocks were executed. This database can then be analyzed with a special application (`tca`) to create reports.

Since coverage analysis is such a powerful tool when used in conjunction with Insure++, it is automatically enabled whenever you compile an application with the `insure` command, unless you specifically disable it, as described in the section “Options used by TCA” on page 194. If you only want to perform coverage analysis (i.e. you don't want the normal Insure++ runtime checking), you can compile and link with the `-Zcov` switch.

**Note:** Please note that if you compile with `insure` and the `-Zcov` option, you must also link with the `-Zcov` option, and vice versa. You cannot use `-Zcov` in only one stage of the build.

## An example - sorting strings

To see this process in action, consider the code shown below, which is a modified version of a bubble sorting algorithm.

```
1: /*
2:  * File: strsort.c
3:  */
4: #include <stdio.h>
5: #include <string.h>
6:
7: bubble_sort(a, n, dir)
8:     char **a;
```

```

9:         int n, dir;
10: {
11:     int i, j;
12:
13:     for(i=0; i<n; i++) {
14:         for(j=1; j<n-i; j++) {
15:             if(dir * strcmp(a[j-1], a[j]) < 0) {
16:                 char *temp;
17:
18:                 temp = a[j-1];
19:                 a[j-1] = a[j];
20:                 a[j] = temp;
21:             }
22:         }
23:     }
24: }
25:
26: main(argc, argv)
27:     int argc;
28:     char **argv;
29: {
30:     int i, dir, length, start;
31:
32:     if (argc > 1 && argv[1][0] == '-') {
33:         if (argv[1][1] == 'a') {
34:             dir = 1; length = argc-2; start = 2;
35:         } else if (argv[1][1] == 'd') {
36:             dir = -1; length = argc-2; start = 2;
37:         }
38:         } else {
39:             dir = 1; length = argc; start = 1;
40:         }
41:     bubble_sort(argv+start, length, dir);
42:     for (i = 2; i < argc; i++)
43:         printf("%s ", argv[i]);
44:     printf("\n");
45:     return 0;
46: }

```

This program sorts a set of strings supplied as command line arguments in either ascending or descending order, according to the settings of the command line switches.

```
strsort -a s1 s2 s3 ...
```

Sorts strings in ascending order.

```
strsort -d s1 s2 s3 ...
```

Sorts strings in descending order.

```
strsort s1 s2 s3 ...
```

Sorts strings in ascending order.

If you wish to try the commands shown in this section, you can use the source code supplied with the Insure++ examples by executing the command

```
cp /usr/local/insure/examples/c/strsort.c .
```

To compile and execute this program with both runtime error detection and coverage analysis enabled, simply use the normal `insure` command

```
insure -g -o strsort strsort.c
```

In addition to compiling the program, this will create a file called `tca.map` which describes the layout of your source file. We can now perform a set of tests on the application. A few samples are shown below. The statements beginning with the `$` symbol are the commands executed and the remaining text is the response of the system.

```
$ strsort -a aaaa bbbb
aaaa bbbb
** TCA log data will be merged with tca.log **

$ strsort -a bbb aaa
aaa bbb
** TCA log data will be merged with tca.log **

$ strsort -d aaa bbbb
bbbb aaa
** TCA log data will be merged with tca.log **

$ strsort -d bbb aaa
bbb aaa
** TCA log data will be merged with tca.log **
```

Note the following features from the example.

- Each time the application is executed, the coverage analysis module issues a message indicating that information is being added to a `log` file, which is created the first time the program is run. This file contains the coverage information for one or more test runs.

- Insure++ issues no error messages during any execution of the program.

## Analyzing the test coverage data

Analysis of the test coverage data is performed using the `tca` command. There are several summary levels. The ones we will introduce here are:

- Overall summary (`default`) - Shows the percentage coverage at the application level - i.e., summed over all program entities.
- Function summary (`-df` switch) - Displays the coverage of each individual function in the application.
- Source code summary (`-ds` switch) - Displays the source code with each block marked as executed (`.`) or not (`!`).

To see these commands in action, execute the command

```
tca tca.log
```

This displays the top level summary, shown below.

```
COVERAGE SUMMARY
=====
1  block untested
12 blocks tested

92% covered
```

As can be seen, the overall coverage is quite high. However, one program block remains untested. To find out which one is untested, execute the command

```
tca -df tca.log
```

This command displays the function level summary, and includes functions that are 100% tested.

```
COVERAGE SUMMARY - by function
=====

blocks  blocks  %cov =  functions sorted
untested tested  %tested  by name
-----
0       4       100%    bubble_sort [strsort.c, line 7-13]
1       8       88%     main        [strsort.c, line 26-45]
```

From this listing, you can see that the function `bubble_sort` is completely tested, but that one block in `main` remains untested. To find out which one, execute the command

```
tca -ds tca.log
```

This results in the following output.

```
UNTESTED BLOCKS - by file
=====
FILE strsort.c 92% covered: 1 untested / 12 tested
/*
* File: strsort.c
*/

...
main(argc, argv)
int argc;
char **argv;
{
int i, dir, length, start;

. ->     if (argc > 1 && argv[1][0] == '-') {
. ->         if (argv[1][1] == 'a') {
. ->             dir = 1; length = argc-2; start = 2;
. ->         } else if (argv[1][1] == 'd') {
. ->             dir = -1; length = argc-2; start = 2;
. ->         }
. ->     } else {
! ->         dir = 1; length = argc; start = 1;
. ->     }
. ->     bubble_sort(argv+start, length, dir);
for (i = 2; i < argc; i++)
. ->     printf("%s ", argv[i]);
. ->     printf("\n");
return 0;
}

...

```

This listing shows exactly which statements have been executed and which have not. The results show that the untested block corresponds to the case where `strsort` is executed with neither the `-a` nor `-d` command line switches.

## Achieving 100% test coverage

The previous analysis tells us that we can achieve 100% test coverage in this example by executing the `strsort` program without either of its two switches.

To complete testing, execute the program with the command

```
strsort aaa bbb
```

This produces the following output

```
[strsort.c:15] **READ_NULL**
>> if(dir * strcmp(a[j-1], a[j]) > 0) {

    Reading null pointer: a[j]

    Stack trace where the error occurred:
        bubble_sort() strsort.c, 15
        main() strsort.c, 41

**Memory corrupted. Program may crash!**

** TCA log data will be merged with tca.log **
```

which indicates that `Insure++` has found an error in this code block. (Finding and fixing this error is left as an exercise for you. Remember that if you built the program with the `-ZCOV` option, this bug would not have been detected by `Insure++`.)

However, when the command `tca tca.log` is executed, the following output is produced

```
COVERAGE SUMMARY
=====
0    blocks untested
13   blocks tested

100% covered
```

which indicates that the application has now been 100% tested.

This means that the set of test cases that have been run, including this last one, completely exercised all the code in this application. It makes sense to incorporate these test cases (in conjunction with `Insure++`) into a quality assurance test suite for this code.

There are several `.psrc` options you can use to control coverage analysis. These are documented in the section “Configuration Options” on page 175 of this manual.

## How to use coverage analysis

If you are using Insure++, coverage analysis information will be automatically built into your program. At any time after you have run your code you can use the `tca` command to find any blocks which have not been executed. For clarity, the process is broken down into three steps.

- Compiling applications and building their coverage analysis database (usually named `tca.map`).
- Running test cases against applications that have been compiled with coverage analysis enabled, which creates entries in the TCA log file (usually named `tca.log`).
- Running the TCA analysis tool to generate coverage analysis reports from the given coverage log file(s).

You can make your program display a coverage analysis report when it exits by adding the

```
insure++.summarize coverage
```

option to your `.psrc` file. The `coverage_switches` option lets you set flags to control the output just as though you were passing those switches to `tca`.

### Step 1: compile time

At compile time, Insure++ creates a database of each file processed, describing how many blocks and statements are in each file and function. This database is called a map file, because it provides TCA with a map of how your program is laid out. By default, the name of this file is `tca.map`, but you can change the name of this file by adding a `coverage_map_file` value to your `.psrc` file.

Ideally, all the files in your application should store their information in the same map file. If your source code is spread across several directories, you will probably want to set the map filename using a full path, e.g.

```
insure++.coverage_map_file ~/project.map
```

If you compile several files simultaneously and they are all trying to modify the same map file, you may end up with a corrupt map file. In this case, you will need to delete the original map file and recompile the application you are interested in.

As mentioned before, if you are only interested in coverage information and not debugging, you can add the `-Zcov` option to the `insure` command lines that build your program. Remember to use `-Zcov` consistently, i.e. at both compilation and linking, if you use it at all.

## Step 2: runtime

At runtime, your program (compiled with Insure++) writes a log file, which records the blocks that were actually executed during a specific run. By default, this file is called `tca.log`, but as with the map file, you can change the name of this file by adding a `coverage_log_file` value to your `.psrc` file. Normally, each time you run your program the new log information will be combined with any found in the existing log file, unless the data is not compatible (because you changed your code and recompiled, for example).

Another useful option is to generate a new log file each time your application runs. You can do this by taking advantage of the `%n filename` option, for example

```
insure++.coverage_log_file tca.log.%n
```

In this example, each run would make a new file, such as `tca.log.0`, `tca.log.1`, and so forth. If your program forks, you will need to use this option so that each child creates its own log file.

## Step 3: using TCA to display information

After you have created one or more log files, you can use the `tca` command to get the information in which you are interested. TCA normally sends its reports to `stdout`. If you would like to use the graphical version to generate coverage reports, please see the section “The TCA Display” on page 169. You can specify any number of log files on the command line, and TCA will combine the data before displaying the results. (If the log files are not compatible, e.g. because they are from

different applications, TCA will throw out the ones that don't match the first log file).

TCA will also need to read the map file created at compilation time. Since this name is stored in the log file, you won't normally need to specify it.

By default, TCA will give you a quick summary of how much of your code was tested. Using different options, you can get detailed reports of coverage by `file`, `function`, or even `block`. For each block, TCA can tell you how many times it was executed, summing over all the log files (unless `coverage_boolean` was on at compile time, the default setting).

Note that if a single statement spans several lines of source code, TCA treats the statement as lying on the last line - this is only important for understanding the output of TCA, and does not effect how coverage statistics are calculated.

## How are blocks calculated?

Unlike some other coverage analysis tools which work on a line-by-line basis, TCA is able to group your code into logical blocks. 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;
```

Some of the advantages of using blocks over lines are,

- 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 detect which paths have been executed in addition to which statements.

Note that conditional expressions containing `&&` or `||` are grouped with the statement they are part of. Also, the three elements of a for loop are treated as part of the `for` statement (e.g. `e1`, `e2`, and `e3` in the code fragment `for(e1;e2;e3)`).

The figures below illustrate these concepts with a simple test program.

This code sample shows how the blocks are determined. In this particular example, there are 16 blocks.

```
#include <ctype.h>

main(int argc, char **argv) {
    int flag;

    if (argc < 2 || !isdigit(argv[1][0])) {
        printf("Bad argument(s)\n");
        exit(1);
    }
    switch(atoi(argv[1])) {
    case 1: case 2: case 3:
        flag = 1;
        break;
    case 4:
    case 5:
        flag = 2;
        break;
    default:
        flag = 0;
        break;
    }
    if (flag > 0) flag = 1; else flag = 0;
    printf("Flag is %\n", flag ? "1" : "0");
    exit(0);
}
```

The next code sample shows the output of `tca -ct -ds tca.log` after several test runs with different values (test ; test 2 ; test 2 ; test 4 ; test 7 ; test 3). By looking at this, you can see which paths have been executed and which have not. Notice that counts are only given at the *beginning* of each block, and not for each statement within each block.

```
BLOCK USAGE - by file
=====
```

```
FILE test.c 87% covered: 2 untested / 14 tested
```

```

        #include <ctype.h>
        main(int argc, char **argv) {
            int flag;
    6 ->     if (argc < 2 || !isdigit(argv[1][0])) {
    1 ->         printf("Bad argument(s)\n");
            exit(1);
    }
        5 ->     switch(atoi(argv[1])) {
    0,2,1 ->     case 1: case 2: case 3:
    3 ->         flag = 1;
            break;
    1 ->     case 4:
    0 ->     case 5:
    1 ->         flag = 2;
            break;
    1 ->     default:
    1 ->         flag = 0;
            break;
    }
    5,4,1 ->     if (flag > 0) flag = 1; else flag = 0;
    5 ->     printf("\nFlag is %s\n", flag ? "1" : "0");
            exit(0);
        }

```

Finally, the next code sample shows the terser output of `tca -ds tca.log`. In this instance, only blocks which have not been executed are marked (corresponding to blocks with a count of zero in the previous figure). The “!” character symbolizes not executed. For lines with multiple blocks, you will also see the “.” character which means that group was executed. This is so you can easily identify which blocks on that line were not tested. Once again, only the first line of code within each block will be marked in this fashion. If you are using the graphical TCA, the first line of the block will be colored as executed (red) while the rest of the block will be colored as not executed (black).

```
UNTESTED BLOCKS - by file
=====
```

```
FILE test.c 87% covered: 2 untested / 14 tested
```

```

#include <ctype.h>

main(int argc, char **argv) {
    int flag;

. ->     if (argc < 2 || !isdigit(argv[1][0])) {
. ->         printf("\`Bad argument(s)\n\`);
            exit(1);
        }
. ->     switch(atoi(argv[1])) {
!.. ->     case 1: case 2: case 3:
. ->         flag = 1;
                break;
. ->     case 4:
! ->     case 5:
. ->         flag = 2;
            break;
. ->     default:
. ->         flag = 0;
            break;
        }
... ->     if (flag > 0) flag = 1; else flag = 0;
. ->     printf("Flag is %s\n", flag ? "1" : "0");
        exit(0)
    }
}
```

# The TCA Display

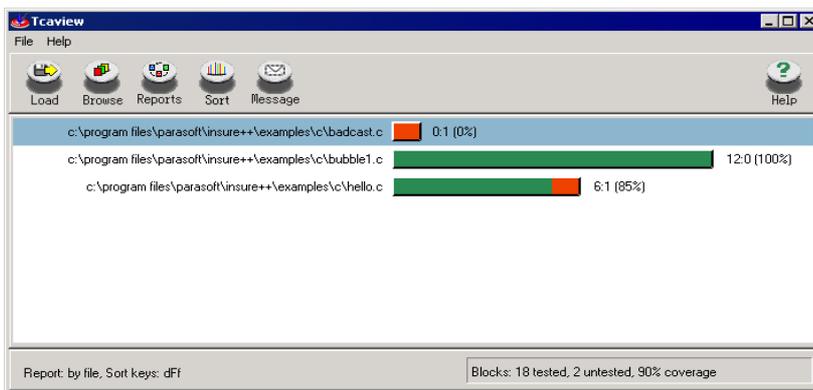
The TCA display is a graphical representation of the reports generated during run time. By utilizing this tool, you will be able to view your `tca.log` files with ease. Much like Insra, TCA allows you to load and save files, browse through source code, and even access on-line help.

The TCA Display may be invoked by calling `tca` with the `-x` switch along with any other command line options. The following subset of TCA command line options are meaningful for the graphical tool, while the remaining unsupported ones are silently ignored.

Command	Explanation
<code>-df</code>	Display by function
<code>-do</code>	Display by object/class
<code>-dF</code>	Display by file
<code>-dd</code>	Display by directory
<code>-ns</code>	Simple function names
<code>-ne</code>	Extended function name - include argument types
<code>-nm</code>	Include modifiers const/volatile in function arguments
<code>-ff name</code>	Only show coverage related to function "name." This option only applies when <code>-df</code> or <code>-dF</code> is also specified.
<code>-fo name</code>	Only show coverage related to object "name." This option applies when <code>-do</code> is also specified.
<code>-fF name</code>	Only show coverage related to file "name." "name" must include the full path to the file. This option only applies when <code>-dF</code> or <code>-df</code> is also specified.
<code>-fd name</code>	Only show coverage related to directory "name." This option only applies when <code>-dd</code> is also specified.
<code>-s {keys}</code>	Sort output by keys (d, F, n, %, #, b, 1)
<code>-ct</code>	Show hit counts in the source browser.

## Loading a report file

By default, TCA displays a report based on the log files that were included in the command line when the program was started. Coverage statistics from additional files may be included in the report by clicking on the **Load** button and selecting a new log file. The data contained in the newly selected log file is combined with the existing data and a new report is generated.



## Browsing the source

The **Browse** button generates a new window containing the next level of coverage detail. For example, if you are currently displaying a report "by directory," clicking **Browse** will open a new window displaying a report "by file" for that directory. If you click **Browse** again, you will get another window displaying a report "by function" for the file(s). Clicking **Browse** a final time displays the source code itself, annotated with coverage information for each block.

Double-clicking on a line in the display is the equivalent to selecting a line and clicking the **Browse** button.

```

Function: main First line: 9
File Help
? Help
Line # Hits # c:\program files\parasoft\insure++\examples\c\slowleak.c
8      #define MAXSLOTS 16
9
10     main()
11     {
12         int i, which;
13         char **pp;
14
15     !   while(1) {
16     !   pp = (char **)malloc(MAXSLOTS*sizeof(char *));
17     !   for(i=0; i<MAXSLOTS; i++) {
18     !   pp[i] = malloc(64+);
19     !   }
20
21     !   Sleep(100);
22
23     /*
24     * Now free them in a random order.....
25     */
26     for(i=0; i<MAXSLOTS; i++) {
27     !   which = (rand() >> 4) % MAXSLOTS;
28     !   if(pp[which]) {
29     !   free(pp[which]);
30     !   pp[which] = 0;
31     !   }

```

## Reports

The level of detail displayed may be changed by clicking on the **Reports** button. A dialog box will appear, allowing you to choose from one of four report types: by directory, by file, by function, or by class.

## Sorting

The order in which the coverage information is presented may be modified by clicking on the **Sort** button. A dialog box will appear in which you can enter the sort keys to be used. Any combination of the following keys may be used.

Sort Keys	Explanation
d	by directory
F	by file name
f	by function
%	by percent covered
#	by number of hits
b	by number of blocks

For example, in order to sort by percent covered and decide collisions by the function name, the sort key string should be %f.

The current sort key string is displayed on the status bar.

## Message

This button becomes active when TCA cannot perform a given task. Clicking it opens a window that describes the error(s).

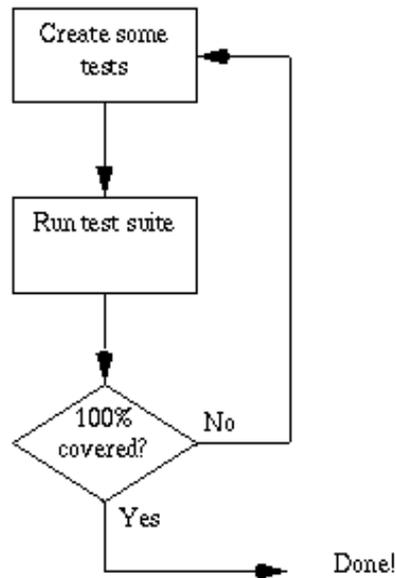
## Help

On-line help can be accessed in two ways. Context-sensitive help can be accessed by clicking on the **Help** button, which causes a question mark cursor to replace the normal arrow, and then clicking on an area of the GUI. If there is help available for that area or button, a window is displayed with information about how to use the area or button that you clicked. You can also access help from the TCA menu bar.

# Building a Test Suite

Now that you have all this coverage analysis information, what's the best way to use it? Typically, you will have several tests for your code designed to exercise various features. Together, these tests make a test suite. After you have run your tests, use TCA to discover which blocks have not been executed. This will indicate deficiencies in your test suite.

At this point, you should create more tests to try and exercise the code that was missed by your test suite so far. After you have created more tests, you can repeat the process. If the goal of 100% coverage is unreachable, you will need to make a subjective decision about how thorough you can afford to be.





# Configuration Options

Insure++ programs read options from files called `.psrc`, which may exist at various locations in the file system. These options control the behavior of Insure++ and programs compiled with Insure++. The files are processed in the order specified below.

- The file `.psrc` in the appropriate lib and compiler subdirectories of the main Insure++ installation directory, e.g.  
`/usr/local/insure/lib.sol/cc/.psrc`  
or  
`/usr/local/insure/lib.rs6000/xlC/.psrc`
- The file `.psrc` in the main installation directory.
- A file `.psrc` in your `$HOME` directory, if it exists.
- A file `.psrc` in the current working directory, if it exists.
- Files specified with the `-zop` switch and individual options specified with the `-zoi` switch to the insure command in the order present on the command line.

In each case, options found in later files override those seen earlier. All files mentioned above will be processed and the options set before any source files are processed. You can also override these options at runtime by using the `_Insure_set_option` function.

Typically, compiler-dependent options are stored in the first location, site-dependent options are stored in the second location, user-dependent options are stored in the third location, and project-dependent options are stored in the fourth location. `-zop` is commonly used for file-dependent options, and `-zoi` is commonly used for temporary options.

## Working on multiple platforms or with multiple compilers

Many projects involve porting applications to several different platforms or the use of more than one compiler. Insure++ deals with this by using two built-in variables, which denote the machine architecture on which you are running and the name of the compiler you are using. Anywhere that

you would normally specify a pathname or filename, you can then use these values to switch between various options, each specific to a particular machine or compiler.

For example, environment variables such as the % notation described in “Filenames” on page 176 are expanded when processing filenames, so the command

```
interface_library $HOME/insure/%a/%c/foo.tqs
```

loads an interface file with a name such as

```
/usr/me/insure/sol/cc/foo.tqs
```

in which the environment variable Home has been replaced by its value and the %a and %c macros have been expanded to indicate the architecture and compiler name in use.

There is one additional comment that must be made here. In the compiler-default .psrc files, there are several interface\_library options of the form

```
Insure++.InterfaceLibrary
$PARASOFT/lib.%a/%c/builtin.tqi \
$PARASOFT/lib.%a/libtqsiic%c.a
```

Despite appearances, the PARASOFT variable used above is not a true environment variable. If the PARASOFT environment variable is not set by the user, it will be expanded automatically by Insure++ itself.

## Option values

The following sections describe the Insure++ configuration options. Options are divided into two classes: compile time and runtime. Modifying one of the compile time options requires that files be recompiled before it can take effect. The runtime options merely require that the program be executed again.

Some options have default values, which are printed in the following section in **boldface**.

## Filenames

A number of the Insure++ options can specify filenames for various configuration and/or output files. You may either enter a simple filename or give a template which takes the form of a string of characters with tokens such as %d, %p, or %V embedded in it. Each of these is expanded to indicate a certain property of your program as indicated in the following tables. The first table lists the options that can be used at both compile and runtime.

Key	Meaning
%a	Machine architecture on which you are running, e.g., sun4, aix4, hp10, etc.
%c	Abbreviated name of the compiler you are using, e.g. cc, gcc, x1C, etc
%r	Insure++ version number, e.g. 6.0
%R	Insure++ version number without periods (.), e.g., version 6.0 becomes 60
%t	.tqs file format version number, e.g., 4.0.0
%T	Similar to %t but with periods (.) removed

This second table lists the tokens available only at runtime.

Key	Meaning
%d	Time of program compilation in format: YYYYMMDDHHMMSS
%D	Time of program execution in format: YYYYMMDDHHMMSS
%n	Integer sufficient to make filename unique, starting at 0
%p	Process I.D.
%v	Name of executable
%V	Directory containing executable

Thus, the name template

```
report_file %v-errs.%D
```

when executed with a program called `foo` at 10:30 a.m. on the 21st of December 2001, might generate a report file with the name

```
foo-errs.20011221103032
```

(The last two digits are the seconds after 10:30 on which execution began.)

You can also include environment variables in these filenames.

## For TCA

Thus, the option

```
coverage_map_file tca.map.%a.%c
```

might generate a report file with the name

```
tca.map.sun4.cc
```

You can also include environment variables in these filenames so that

```
$HOME/tca/tca.map.a%c%
```

generates the same filename as the previous example, but also ensures that the output is placed in the `tca` sub-directory of the user's home directory.

## Options used by Insure++

### Compiling/linking

```
insure++.auto_expand [off|on]
```

Specifies how Insure++ should treat suspected “stretchy” arrays. See “Stretchy arrays” on page 76 for a description of stretchy arrays, and the table below for explanations of the allowed keywords for this option. Multi-

dimensional arrays are never automatically expanded. To tell Insure++ that a specific array is stretchy, use the `expand` option (see page 184).

Keyword	Meaning
<code>all</code>	All arrays at the end of structs, classes, and unions are treated as stretchy, regardless of size
<code>off</code>	No automatic detection of stretchy arrays
<code>on</code>	If the last field of a struct, class, or union is an array and has no size, size 0, or size 1, it is treated as stretchy. Note that only some compilers allow 0 or empty sizes, but size 1 is very common for stretchy arrays

`insure++.c_as_cpp [on|off]`

Specifies whether files with the `.c` extension should be treated as C++ source code. With this option `off`, Insure++ will treat files with the `.c` extension as C code only. If you use C++ code in `.c` files, you should turn this option `on`.

`insure++.checking_uninit [on|off]`

Specifies that the code to perform flow-analysis and check for uninitialized variables should not be inserted. Runtime uninitialized variable checking is then limited to uninitialized pointer variables. See page 188 for the runtime effects of this option.

`compiler compiler_name`

Specifies the name of an alternative compiler, such as `gcc`. This option overrides all other `compiler_*` options: `compiler_c`, `compiler_cpp`, and `compiler_default`. The indicated compiler will be called every time `insure++` is called.

`compiler_c C_compiler_name`

Specifies the name of the default C compiler. This compiler will be called for any `.c` files. The default is `cl`. This option is overridden by the `compiler` and `compiler_acronym` options.

`compiler_cpp C++_compiler_name`

Specifies the name of the default C++ compiler, such as `cl`. This compiler

will be called for any `.cc`, `.cpp`, and `.cxx`. The default is platform-dependent. This option is overridden by the `compiler` and `compiler_acronym` options.

`compiler_default` [`c`|`cpp`]

Specifies whether the default C or C++ compiler should be called to link when there are no source files on the link line. This option is overridden by the `compiler` and `compiler_acronym` options.

`compiler_deficient` [`all`|`address`|`cast`|`enum`|`member_pointer`|`scope_resolution`|`static_temps`|`struct_offset`|`types`|`no_address`|`no_cast`|`no_enum`|`no_member_pointer`|`no_scope_resolution`|`no_static_temps`|`no_struct_offset`|`no_types`|`none`]

Specifies which features are not supported by your compiler. The default is compiler-dependent.

Keyword	Meaning
<code>all</code>	Includes all positive keywords
<code>address/no_address</code>	
<code>cast/no_cast</code>	
<code>enum/no_enum</code>	
<code>member_pointer/no_member_pointer</code>	
<code>scope_resolution/no_scope_resolution</code>	
<code>static_temps/no_static_temps</code>	
<code>struct_offset/no_struct_offset</code>	
<code>types/no_types</code>	
<code>none</code>	Compiler handles all cases

Different compilers require different levels of this option as indicated in the compiler-specific `README` files and in `($PARASOFT)//$compiler`.

`compiler_fault_recovery [off|on]`

This option controls how Insure++ recovers from errors during compilation and linking. With fault recovery `on`, if there is an error during compilation, Insure++ will simply compile with the compiler only and will not process that file. If there is an error during linking, Insure++ will attempt to take corrective action by using the `-Zs1` option. If this option is turned `off`, Insure++ will make only one attempt at each compile and link.

`compiler_keyword [*|const|inline|signed|volatile] keyword`  
 Specifies a new compiler keyword (by using the `*`) or a different name for a standard keyword. For example, if your compiler uses `__const` as a keyword, use the option

```
compiler_keyword const __const
```

`compiler_options keyword value`

Specifies various capabilities of the compiler in use, as described in the following table.

Keyword	Value	Meaning
<code>ansi</code>	None	Assumes compiler supports ANSI C (default)
<code>bfunc &lt;type&gt;</code>	Function name	Specifies that the given function is a “built-in” that is treated specially by the compiler. The optional <code>type</code> keyword specifies that the built-in has a return type other than <code>int</code> . Currently, only <code>long</code> , <code>double</code> , <code>char *</code> , and <code>void *</code> types are supported.
<code>btype</code>	Type name	Specifies that the given type is a “built-in” that is treated specially by the compiler
<code>bvar &lt;type&gt;</code>	Variable name	Specifies that the given variable is a “built-in” that is treated specially by the compiler. The optional <code>type</code> keyword specifies that the built-in has a return type other than <code>int</code> . Currently, only <code>long</code> , <code>double</code> , <code>char *</code> , and <code>void *</code> types are supported.

Keyword	Value	Meaning
<code>esc_x</code>	Integer	Specifies how the compiler treats the <code>\x</code> escape sequence. Possible values are 0 treat <code>\x</code> as the single character <code>x</code> (Kernighan & Ritchie style) -1 treat as a hex constant. Consume as many hex digits as possible >0 treat as a hex constant. Consume at most the given number of hex digits
<code>for_scope</code>	nested notnested optional	Specifies how <code>for(int i; ...; ...)</code> is scoped. Possible values are: nested New ANSI standard, always treat as nested. notnested Old standard, never treat as nested. optional New standard by default, but old-style code is detected and treated properly (and silently)
<code>knr</code>	None	Assumes compiler uses Kernighan and Ritchie (old-style) C
<code>loose</code>	None	Enables non-ANSI extensions (default)
<code>namespaces</code>	None	Specifies that <code>namespace</code> is a keyword (default)
<code>nonamespaces</code>	None	Specifies that <code>namespace</code> is not a keyword
<code>promote_long</code>	None	Specifies that integral data types are promoted to <code>long</code> in expressions, rather than <code>int</code>
<code>sizeof</code>	d, ld, u, lu	Specifies the data type returned by the <code>sizeof</code> operator, as follows: d=int, ld=long, u=unsigned int, lu=unsigned long.

Keyword	Value	Meaning
<code>strict</code>	None	Disables non-ANSI extensions (compiler dependent)
<code>xfunctype</code>	Function name	Indicates that the named function takes an argument which is a data type rather than a variable (e.g., <code>alignof</code> )

`coverage_map_data` [`on`|`off`]

Prompts Insure++ to generate coverage map data for TCA.

`coverage_map_file` [`filename`]

Specifies the full path to the directory where Insure++ writes the `tca.map` file.

`coverage_only` [`on`|`off`]

Compiles and generates coverage information (`tca.map`). It also compiles and links source code for TCA.

`error_format` `string`

Specifies the format for error message banners generated by Insure++. The string argument will be displayed as entered with the macro substitutions taking place as shown in the following table. The string may also contain standard C formatting characters, such as `\n`. (For examples, see “Customizing the output format” on page 69)

Key	Expands to
<code>%c</code>	Error category (and sub-category if required)
<code>%d</code>	Date on which the error occurs ( <code>DD-MM-YYYY</code> )
<code>%f</code>	Filename containing the error
<code>%F</code>	Full pathname of the file containing the error
<code>%h</code>	Name of the host on which the application is running
<code>%l</code>	Line number containing the error
<code>%p</code>	Process ID of the process incurring the error

Key	Expands to
<code>%t</code>	Time at which the error occurred (HH:MM:SS)

`expand subtypename`

Specifies that the named structure element is “stretchy”. See “Stretchy arrays” on page 76 for a description of stretchy arrays. See also the `auto_expand` option on page 178 for details on automatic detection and handling of stretchy arrays.

`file_ignore string`

Specifies that any file which matches the string will not be processed by Insure++, but will be passed straight through to the compiler. The string should be a glob-style regular expression.

This option allows you to avoid processing files that you know are correct. This can significantly speed up execution and shrink your code.

`function_ignore file::function_name`

This option tells Insure++ not to instrument the given function (the file qualifier is optional). This is equivalent to turning off the checking for that routine. If the function in question is a bottle-neck, this may dramatically increase the runtime performance of the code processed with Insure++. `function_name` can now (version 3.1 and higher) accept the `*` wildcard.

For example, the option

```
function_ignore foo*
```

turns off instrumentation for the functions `foo`, `foobar`, etc..

`header_ignore string`

Specifies that any function in the filename specified by the string will not be instrumented by Insure++. The string should be a glob-style regular expression and should include the full path.

This option allows you to avoid doing runtime checking in header files that you know are correct. This can significantly speed up execution and shrink your code. Please note, however, that the file must still be parsed by Insure++, so this option will not eliminate compile-time warnings and errors, only runtime checking.

`init_extension [c|cc|C|cpp|cxx|c++]`

This option tells Insure++ to use the given extension and language for the

Insure++ initialization code source file. The extension can be any one of the Insure++-supported extensions: `c` (for C code) or `cc`, `cpp`, `cxx`, or `c++` (for C++ code). This option only needs to be used to override the default, which is the extension used by any source files on the `insure++` command line. If there are no source files on the command line, e.g., a separate link command, Insure++ will use a `c` extension by default.

`linker linker_name`

Specifies the name of an alternative linker. This only applies if you are using the `inslink` command.

`linker_source source_code`

This option tells Insure++ to add the given code to its initialization file. This can help eliminate unresolved symbols caused by linker bugs.

`linker_stub symbol_name`

This option tells Insure++ to create and link in a dummy function for the given `symbol_name`. This can help eliminate unresolved symbols caused by linker bugs.

`malloc_replace [on|off]`

If `on`, Insure++ links its own version of the dynamic memory allocation libraries. This gives Insure++ additional error detection abilities, but may have different properties than the native library (for example, it will probably use more memory per block allocated). Setting this option to `off` links in the standard library. This option does not apply in LRT mode.

`pragma_ignore string`

Any pragma which matches the string will be deleted by Insure++. The string should be a glob-style regular expression.

`rename_files [on|off]`

Normally, Insure++ creates an intermediate file which is passed to the compiler. In some cases, this may confuse debuggers. If this is the case, you can set this option and Insure++ will then rename the files during compilation so that they are the same. In this case, an original source file called `foo.c` would be renamed `foo.c.ins_orig` for the duration of the call to Insure++.

`report_banner [on|off]`

Controls whether or not a message is displayed on your terminal, reminding you that error messages have been redirected to a file. (See “The report file” on page 67)

`report_file [filename|insra|stderr]`

Specifies the name of the report file. Environment variables and various pattern generation keys may appear in `filename` (see “Filenames” on page 176). Use of the special filename `insra` tells Insure++ to send its output to Insra.

`sizeof type value`

This option allows you to specify data type sizes which differ from the host machine, which is often necessary for cross compilation. `value` should be the number `sizeof(type)` would return on the target machine. Allowed `type` arguments are `char`, `double`, `float`, `int`, `long`, `long double`, `long long`, `short`, and `void *`.

`stack_internal [on|off]`

If you are using the `symbol_table off` runtime option (see page 193), you can set this option to `on` and recompile your program to get filenames and line numbers in stack traces without using the symbol table reader.

`stdlib_replace [on|off]`

Links with an extra Insure++ library that checks common function calls without requiring recompilation. This is useful for finding bugs in third-party libraries or for quickly checking your program without fully recompiling with Insure++. This option does not apply in LRT mode.

`string_table [on|off]`

Moves the string table into a separate file.

`suppress code`

Suppresses compile time messages matching the indicated error code. Context-sensitive suppression does not apply at compile time (see “Suppressing error messages” on page 73).

`suppress_output string`

Suppresses compile time messages including the indicated error string (see “Suppressing other warning messages” on page 75). For example, to suppress the warning:

```
[foo.c:5] Warning: bad conversion in assignment: char * = int *
>> ptr = iptr;
```

add the following string to this value:

```
bad conversion in assignment
```

`suppress_warning code`

Suppresses C++-specific compile time messages matching the indicated

warning code (see “Suppressing other warning messages” on page 75). `code` should match the numerical code Insure++ prints along with the warning message you would like to suppress. The codes correspond to the chapter, section, and paragraph(s) of the draft ANSI standard on which the warning is based. For example, to suppress the warning:

```
Warning:12.3.2-5: return type may not be specified for conversion
functions
```

add the following string to this value.

```
12.3.2-5
```

```
temp_directory path
```

Specifies the directory where Insure++ will write its temporary files, e.g. `C:\tmp`. The default is the Windows temporary directory. Setting `path` to a directory local to your machine can dramatically improve compile-time performance if you are compiling on a remotely mounted file system.

```
threaded_runtime [on|off]
```

Specifies which Insure++ runtime library will be used at link time. This option should be turned on before linking threaded programs with Insure++.

```
uninit_flow [1|2|3|...|100|...|1000]
```

When Insure++ is checking for uninitialized memory, a lot of the checks can be deduced as either correct or incorrect at compile time. This value specifies how hard Insure++ should try to analyze this at compile time. A high number will make Insure++ run slower at compile time, but will produce a faster executable. Values over 1000 are not significant except for very complicated functions.

```
unsuppress code
```

Enables compile time messages matching the indicated error code. Context sensitive suppression is not supported at compile time (see “Enabling error messages” on page 76).

```
virtual_checking [on|off]
```

Specifies whether `VIRTUAL_BAD` error messages will be generated. See “VIRTUAL\_BAD” on page 204 for more information about this error message.

## Running

`checking_uninit` [**on**|**off**]

If set to `off`, this option specifies that the code to perform flow-analysis and checking for uninitialized variables should not be executed, if present. See “Compiling/linking” on page 178 for the compile time effects of this option. Runtime uninitialized variable checking is then limited to uninitialized pointer variables.

`checking_uninit_min_size` [1|2|3|...]

Specifies the minimum size in bytes of data types on which Insure++ should perform full uninitialized memory checking. The default is 2, which means that `chars` will not be checked by default. Setting this option to 1 will check `chars`, but may result in false errors being reported. These can be eliminated by using the `checking_uninit_pattern` option to change the pattern used (see below).

`checking_uninit_pattern` `pattern`

Specifies the pattern to be used by the uninitialized memory checking algorithm. The default is `deadbeef`. `pattern` must be a valid, 8-digit hexadecimal value.

`coverage_banner` [**on**|**off**]

Prompts Insure++ to display a message about coverage information written to `tca.log`.

`coverage_log_file`

Specifies the full path to the directory where Insure++ writes the `tca.log` file.

`coverage_map_file` [`filename`]

Specifies the full path to the directory where Insure++ writes the `tca.map` file.

`coverage_overwrite` [**on**|**off**]

Determines if the `tca.log` file will be overwritten on each run.

`demangle` [**off**|**on**|`types`|`full_types`]

Specifies the level of function name demangling in reports generated by Insure++. If you have a function

```
void func(const int)
```

you will get the following results:

Keyword	Result
<code>off</code>	<code>func__FCi</code>
<code>on</code>	<code>func</code>
<code>types</code>	<code>func(int)</code>
<code>full_types</code>	<code>func(const int)</code>

`error_format` string

Specifies the format for error message banners generated by Insure++. The string argument will be displayed as entered with the macro substitutions taking place as shown in the table on page 183. The string may also contain standard C formatting characters, such as `\n`. (For examples, see “Customizing the output format” on page 69.)

`exit_hook` [`on|off`]

Normally, Insure++ uses the appropriate `atexit`, `onexit`, or `on_exit` function call to perform special handling at exit. If for some reason, this is a problem on your system, you can disable this functionality via the `exit_hook` option.

`exit_on_error` [`0|1|2|3|...`]

Causes the user program to quit (with non-zero exit status) after reporting the given number of errors. The default is 0, which means that all errors will be reported and the program will terminate normally.

`exit_on_error_banner` [`on|off`]

Normally, when Insure++ causes your program to quit due to the `exit_on_error` option, it will print a brief message like the following:

```
** User selected maximum error count reached: 10. Program exiting.**
```

Setting this option to `off` will disable this message.

`free_delay` [`0|1|2|3|...|119|...`]

This option controls how long the Insure++ runtime holds onto `free`'d blocks before allowing them to be reused. This is not necessary for error detection, but can be useful in modifying the behavior of your program for stress-testing. The number represents how many freed blocks are held

back at a time--large numbers limit memory reuse, and 0 maximizes memory reuse. Please note that this option is only active if `malloc_replace` was on during linking.

`free_pattern pattern`

Specifies a pattern that will be written on top of memory whenever it is freed. This pattern will be repeated for each byte in the freed region (this option is available only if `malloc_replace` was on at compile time). The default is 0, which means no pattern will be written.

**Note:** On some systems whose libraries assume freed memory is still valid, this may cause your program to crash.

`ignore_wild [on|off]`

Specifies whether Insure++ will do checking for wild pointers. Turning this option on turns off wild pointer checking.

`leak_combine [none|trace|location]`

Specifies how to combine leaks for the memory leak summary report. Combining by `trace` means all blocks allocated with identical stack traces will be combined into a single entry. Combining by `location` means all allocations from the same file and line (independent of the rest of the stack trace) will be combined. `none` means each allocation will be listed separately.

`leak_search [on|off]`

Specifies additional leak checking at runtime before a leak is reported. Requires that the symbol table reader be turned on. If “full” is selected, any pointer to anywhere in a block is sufficient to consider it not leaked. Selecting the “full” option is useful when getting false leaks from 3rd party libraries.

`leak_sort [none|frequency|location|size]`

Specifies by what criterion the memory leak summary report is sorted. Setting this to `none` may provide better performance at exit if you have many leaks.

`leak_summary_filter`

Controls which blocks are reported in the “leaks detected at exit” and “outstanding” sections of the leak summary. For example,

```
leak_summary_filter *main
```

Restricts the leak summary to those blocks with stack traces ending in `main`.

`leak_sweep` [`on`|`off`]

Specifies additional leak checking at the termination of the program. Requires that the symbol table reader be turned on. If “full” is selected, any pointer to anywhere in a block is sufficient to consider it not leaked. Selecting “full” is useful when getting false leaks from 3rd party libraries. Leaks detected will be reported using the `summarize_leaks` option (see page 192).

`leak_trace` [`on`|`off`]

This option determines whether or not full stack traces will be shown in the memory leak summary report.

`new_overhead` [`0`|`2`|`4`|`6`|`8`|`...`]

Specifies the number of bytes allocated as overhead each time `new[]` is called. The default is compiler-dependent, but is typically 0, 4, or 8.

`pointer_slack` [`0`|`1`|`2`]

This controls a heuristic in `Insure++`. When a pointer does not point to a valid block, but does point to an area 1 byte past the end of a valid block, does the pointer really point to that block? The value of this argument controls `Insure++`'s answer. The default should be changed only if `Insure++` is not working correctly on your program.

Value	Meaning
0	Never assume the pointer points to the previous block
1	Assume the pointer points to the previous block if that block was dynamically allocated
2	Always assume the pointer points to the previous block. This tends to be incorrect for stack and global variables, since they are usually adjacent in memory

`report_banner [on|off]`

Controls whether or not a message is displayed on your terminal, reminding you that error messages have been redirected to a file (see “Filenames” on page 176).

`report_file [filename|Insr|stderr]`

Specifies the name of the report file. Environment variables and various pattern generation keys may appear in `filename` (see “Filenames” on page 176). Use of the special filename `Insr` tells Insure++ to send its output to `Insr`.

`report_limit [-1|0|1|2|3|...]`

Displays only the first given number of errors of each type at any particular source line. Setting this option to `-1` will show all errors. Setting it to `0` will only show errors in summary reports, and not at runtime. (See “Displaying repeated errors” on page 70)

`report_overwrite [on|off]`

If set to `off`, error messages are appended to the report file rather than overwriting it on each run.

`source_path dir1 dir2 dir3`

This option takes a list of directories in which to search for source files (see “Searching for source code” on page 72). This will only be necessary if your source code has moved since it was compiled, as Insure++ remembers where all your source files are located.

`spy_disable key`

This option tells Insure++ not to use the interface specified by the key. The interfaces will then not be inserted during instrumentation at compile-time. For example,

```
spy_disable ole32.dll.
```

`summarize [bugs] [coverage] [leaks] [outstanding]`

Generates a summary report of errors (see “Report summaries” on page 77), memory leaks (see “The leak summaries” on page 80), outstanding allocated memory blocks, or coverage analysis (see “The coverage summary” on page 81). In the latter case, the `coverage_switches` option (see “Options used by TCA” on page 194) is consulted to decide how to present coverage data. The `leaks` and `outstanding` reports are affected by the `leak_combine`, `leak_sort`, and `leak_trace` options. With no arguments, this option will summarize the `bugs` and `leaks` summaries. This option has changed slightly in

versions 3.1 and higher. The old leak defaults are equivalent to `leak_combine location, leak_sort location, leak_trace off`. The old detailed option is replaced by `leak_trace on`.

`summarize_on_error [0|1|2|3|...]`

Specifies how many errors must be generated before a summary (if requested) is printed. The default is 0, which means that summaries are always printed on demand. If the number is 1 or higher, summaries are only printed if *at least* the given number of bugs (or leaks) occurs. Suppressed errors do not count towards this number. If no argument is given with this option, a value of 1 is assumed.

`suppress code [{context}]`

Suppress error messages matching the given error code and occurring in the (optionally) specified context. (See “Suppressing error messages” on page 73.)

`symbol_banner [on|off]`

If set, Insure++ displays a message indicating that the program’s symbol table is being processed whenever an application starts.

`symbol_sizes_ignore`

This option tells Insure++ not to read variable size information from the specified library.

`symbol_table [on|off]`

If set to `on`, Insure++ will read the executable symbol table at startup. This enables Insure++ to generate full stack traces for third party libraries as well as for code compiled with Insure++. If this option is turned off, the stack traces will show only functions compiled with Insure++, but the application will use less dynamic memory and be faster on startup. To get filenames and line numbers in stack traces with this option off, you must compile your program with the `stack_internal on` option. (See “Compiling/linking” on page 178.)

`trace [on|off]`

Turns program tracing on and off. In order to get file names and line numbers in the trace output, you must have the `stack_internal on` option set when compiling the program. See “Tracing” on page 104 for more information about program tracing.

`trace_banner [on|off]`

Specifies whether to print message at runtime showing file to which the trace output will be written.

`trace_file [filename|stderr]`

Specifies the name of the file to which the trace output will be written. `filename` may use the same special tokens shown on “Filenames” on page 176.

`unsuppress code [{context}]`

Enables error messages matching the given error codes and occurring in the (optionally) specified context. (See “Suppressing error messages” on page 73.)

## Options used by Insra

`visual [editor command]`

Specifies how Insra should call an editor to display the line of source code causing the error. Insra will expand the `%l` token to the line number and the `%f` token to the file name before executing the given command. It is important to include the full path of any binary that lives in a location not on your path. Setting this option with no command string disables source browsing from Insra.

## Options used by TCA

The coverage analysis process consists of the following three stages:

- Compiling applications with Insure++ and building their coverage analysis database (usually named `tca.map`).
- Running test cases against applications that have been compiled with coverage analysis enabled, which creates entries in the TCA log file (usually named `tca.log`).
- Running the `tca` analysis tool to see the coverage analysis results.

The sections below each describe options appropriate to one of these stages.

## Compiling

`insure++.coverage_boolean [on|off]`

If set to `on`, the only data that will be stored is whether or not each block was executed. If `off`, the number of times each block was executed is also recorded. Setting this option to `on` will cause your program to compile and run slightly faster.

`insure++.coverage_map_data [on|off]`

If set to `on`, coverage analysis data is collected whenever applications are compiled. Such applications are then candidates for collecting coverage analysis data at runtime. Setting this option to `off` disables this. Applications must be compiled with this option on before the runtime coverage analysis options have any effect.

`insure++.coverage_map_file filename`

Specifies the name of the file to which the coverage analysis database will be written. Filenames may be specified using any of the standard methods that make sense at compile time. (See “Filenames.”) For example, you cannot use `%P` or `%D` with this option. If this option is not specified, the default filename `tca.map` is used.

## Running

`insure++.coverage_banner [on|off]`

If set to `on`, a message is displayed at runtime indicating the file to which coverage analysis data will be written. Setting this to `off` disables this message.

`insure++.coverage_log_data [on|off]`

If set to `on`, coverage analysis data is collected whenever applications which have been compiled for coverage analysis are executed. Setting this option to `off` disables this.

`insure++.coverage_log_file filename`

Specifies the name of the file to which coverage analysis data will be written. Filenames may be specified using any of the standard methods. (Refer to “Filenames” on page 176.) If this option is not specified, the default filename `tca.log` is used.

`insure++.coverage_overwrite [on|off]`

Indicates how data from successive application runs will be merged with

any existing data. If `on` the existing log file will be overwritten each time the application runs. If this is turned `off`, then each run causes new data to be added to the existing log file to form a new, combined result. In this mode, the log file data will still be discarded if the executable has changed since the last recorded log data.

`insure++.coverage_switches switches`

Specifies the command line arguments to be passed to the `tca` command when it is executed as a result of a “`summarize coverage`” option.

## Running TCA

`registertool TCA version`

Used for internal maintenance. This option should not be modified.

`tca.password arg1 arg2 arg3`

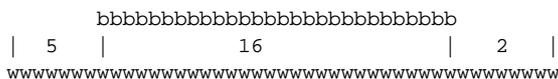
Used for internal maintenance. This option should not be added or modified by hand. Licenses should be managed with `pslic`.



written. The block being written is longer than the actual memory block, which causes the error.

The numbers shown indicate the size, in bytes, of the various regions and match those of the textual error message.

The relative length and alignment of the rows of characters is intended to indicate the size and relative positioning of the memory blocks which cause the error. The above case shows both blocks beginning at the same position with the written block extending beyond the end of the memory region. If the region being written extended both before and after the available block, a diagram such as the following would have been displayed.



Completely disjointed memory blocks are indicated by a diagram of the form



Similar diagrams appear for both `READ_OVERFLOW` and `WRITE_OVERFLOW` errors. In the former case, the block being read is represented by a row of `r` characters instead of `w`'s. Similarly, the memory regions involved in parameter size mismatch errors are indicated using a row of `p` characters for the parameter block. (See "PARM\_BAD\_RANGE" on page 287)

Reference

# Error Codes

This section is intended to provide a reference for the various error messages generated by Insure++. It consists of two parts.

The first lists each error code alphabetically together with its interpretation and an indication of whether or not it is suppressed by default. The second gives a detailed description of each error including:

- A brief explanation of what problem has been detected.
- An example program that generates a similar error.
- Output that would be generated by running the example, with annotations indicating what the various pieces of the diagnostic mean and how they should be interpreted in identifying your own problems.

Note that the exact appearance of the error message might depend heavily on how Insure++ is currently configured.

- A brief description of ways in which the problem might be eliminated.

Note that sometimes you will see values identified as `<argument #>` or `<return>` instead of names from your program. In this case, `<argument n>` refers to the *n*th argument passed to the current function (i.e. the one where the error was detected), and `<return>` refers to a value returned from the function indicated.

Code	Meaning	Enabled?
ALLOC_CONFLICT	Mixing malloc/free with new/delete	Y
(badfree)	Free called on block allocated with new	Y
(baddelete)	Delete called on block allocated with malloc	Y
BAD_CAST	Cast of pointer loses precision	Y
BAD_DECL	Incompatible global declarations	Y
BAD_FORMAT	Mismatch in format specification	N
(sign)	int vs. unsigned int	N
(compatible)	int vs. long, both same size	N
(incompatible)	int vs. double	Y
(other)	Wrong number of arguments	Y
BAD_INTERFACE	Declaration of function in interface conflicts with declaration in program	Y
BAD_PARM	Mismatch in argument type	N
(sign)	int vs. unsigned int	N
(compatible)	int vs. long, both same size	N
(incompatible)	int vs. double	Y
(pointer)	All pointers are equivalent	Y
(union)	Require exact match on unions	Y
(other)	Wrong number of arguments	Y

Code	Meaning	Enabled?
COPY_BAD_RANGE	Attempt to copy out-of-range pointer	N
COPY_DANGLING	Attempt to copy dangling pointer	N
COPY_UNINIT_PTR	Attempt to copy uninitialized pointer	N
COPY_WILD	Attempt to copy wild pointer	N
DEAD_CODE	Code is not evaluated, has no effect, or is unreachable	N
(emptyloopbody)	Loop body is empty	N
(emptystmt)	Statement is empty	N
(noeffect)	Code has no effect	N
(notevaluated)	Code is not evaluated	N
DELETE_MISMATCH	Mismatch between <code>new/new[]</code> and <code>delete/delete[]</code>	N
(bracket)	<code>new, delete[]</code>	Y
(nobracket)	<code>new[], delete</code>	Y
EXPR_BAD_RANGE	Expression exceeded range	N
EXPR_DANGLING	Expression uses dangling pointer	N
EXPR_NULL	Expression uses <code>NULL</code> pointer	Y
EXPR_UNINIT_PTR	Expression uses uninitialized pointer	Y
EXPR_UNRELATED_PTRCMP	Expression compares unrelated pointers	Y

Code	Meaning	Enabled?
EXPR_UNRELATED_PTRDIFF	Expression subtracts unrelated pointers	Y
EXPR_WILD	Expression uses wild pointer	N
FREE_BODY	Freeing memory block from body	Y
FREE_DANGLING	Freeing dangling pointer	Y
FREE_GLOBAL	Freeing global memory	Y
FREE_LOCAL	Freeing local memory	Y
FREE_UNINIT_PTR	Freeing uninitialized pointer	Y
FREE_WILD	Freeing wild pointer	Y
FUNC_BAD	Function pointer is not a function	Y
FUNC_NULL	Function pointer is NULL	Y
FUNC_UNINIT_PTR	Function pointer is uninitialized	Y
INSURE_ERROR	Internal error	Y
INSURE_WARNING	Output from <code>iic_warning</code>	N
LEAK_ASSIGN	Memory leaked due to pointer reassignment	Y
LEAK_FREE	Memory leaked freeing block	Y
LEAK_RETURN	Memory leaked by ignoring return value	Y
LEAK_SCOPE	Memory leaked leaving scope	Y
PARAM_BAD_RANGE	Array parameter exceeded range	Y

Code	Meaning	Enabled?
PARAM_DANGLING	Array parameter is dangling pointer	Y
PARAM_NULL	Array parameter is NULL	Y
PARAM_UNINIT_PTR	Array parameter is uninitialized pointer	Y
PARAM_WILD	Array parameter is wild	Y
READ_BAD_INDEX	Reading array out of range	Y
READ_DANGLING	Reading from a dangling pointer	Y
READ_NULL	Reading NULL pointer	Y
READ_OVERFLOW		N
(normal)	Reading overflows memory	Y
(nonnull)	String is not NULL-terminated within range	Y
(string)	Alleged string does not begin within legal range	Y
(struct)	Structure reference out of range	Y
(maybe)	Dereferencing structure of improper size (may be o.k.)	N
READ_UNINIT_MEM	Reading uninitialized memory	N
(copy)	Copy from uninitialized region	N
(read)	Use of uninitialized value	Y
READ_UNINIT_PTR	Reading from uninitialized pointer	Y

Code	Meaning	Enabled?
READ_WILD	Reading wild pointer	Y
RETURN_DANGLING	Returning pointer to local variable	Y
RETURN_FAILURE	Function call returned an error	N
RETURN_INCONSISTENT	Function returns inconsistent value	N
(level 1)	No declaration, returns nothing	N
(level 2)	Declared <code>int</code> returns nothing	Y
(level 3)	Declared non- <code>int</code> , returns nothing	Y
(level 4)	Returns different types at different statements	Y
UNUSED_VAR	Unused variables	N
(assigned)	Assigned but never used	N
(unused)	Never used	N
USER_ERROR	User generated error message	Y
VIRTUAL_BAD	Error in runtime initialization of virtual functions	Y
WRITE_BAD_INDEX	Writing array out of range	Y
WRITE_DANGLING	Writing to a dangling pointer	Y
WRITE_NULL	Writing to a <code>NULL</code> pointer	Y
WRITE_OVERFLOW		N
(normal)	Writing overflows memory	Y

Code	Meaning	Enabled?
(struct)	Structure reference out of range	Y
(maybe)	Dereferencing structure of improper size (may be o.k.)	N
WRITE_UNINIT_PTR	Writing to an uninitialized pointer	Y
WRITE_WILD	Writing to a wild pointer	Y

# ALLOC\_CONFLICT

## Memory allocation conflict

This error is generated when a memory block is allocated with `new` (`malloc`) and freed with `free` (`delete`).

Insure++ distinguishes between the two possibilities as follows:

- `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 do allow this, but it is not good programming practice and could be a portability problem.

## Problem #1

The following code shows a typical example of allocating a block of memory with `new` and then freeing it with `free`, instead of `delete`.

```

1:      /*
2:      * File: alloc1.cpp
3:      */
4:      #include <stdlib.h>
5:
6:      int main() {
7:          char *a;
8:
9:          a = new char;
10:         free(a);
11:         return 0;
12:     }
```

## Diagnosis (at runtime)

```

1 [alloc1.cpp:10] **ALLOC_CONFLICT**
  >>                free(a);

2     Memory allocation conflict: a

3     free() used to deallocate memory which was allocated
```

```

        using new
        a, allocated at:
        main()alloc1.cpp, 9

4 Stack trace where the error occurred:
    main()alloc1.cpp, 10

```

1. Source line at which the problem was detected.
2. Brief description of the problem.
3. Description of the conflicting allocation/deallocation.
4. Stack trace showing the function call sequence leading to the error.

## Problem #2

The following code shows another typical example of this type of 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:     }

```

## Diagnosis (at runtime)

```

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

2     Memory allocation conflict: a

3     delete operator used to deallocate memory not
      allocated by new
      block allocated at:

```

## ALLOC\_CONFLICT

```
malloc()(interface)
main()alloc2.cpp, 9
```

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

1. Source line at which the problem was detected.
2. Brief description of the problem.
3. Description of the conflicting allocation/deallocation.
4. Stack trace showing the function call sequence leading to the error.

### Repair

This type of error can be corrected by making sure that all your memory allocations match up.

# BAD\_CAST

## Cast of pointer loses precision

Porting code between differing machine architectures can be difficult for many reasons. A particularly tricky problem occurs when the sizes of data objects, particularly pointers, differ from that for which the software was created. This error occurs when a pointer is cast to a type with fewer bits, causing information to be lost, and is designed to help in porting codes to architectures where pointers and integers are of different lengths.

Note that compilers will often catch this problem unless the user has “carefully” added the appropriate typecast to make the conversion “safe”.

## Problem

The following code shows a pointer being copied to a variable too small to hold all its bits.

```

1:      /*
2:      * File: badcast.c
3:      */
4:      main()
5:      {
6:          char q, *p;
7:
8:          p = "Testing";
9:          q = (char)p;
10:         return 0;
11:     }
```

## Diagnosis (during compilation)

```

1 [badcast.c:9] **BAD_CAST**
2   Cast of pointer loses precision: (char) p
>>   q = (char) p;
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.

## Repair

This error normally indicates a significant portability problem that should be corrected by using a different type to save the pointer expression. In ANSI C the type `void *` will always be large enough to hold a pointer value.

# BAD\_DECL

## Global declarations are inconsistent

This error is generated whenever Insure++ detects that a variable has been declared as two different types in distinct source files. This can happen when there are two conflicting definitions of an object or when an `extern` reference to an object uses a different type than its definition.

In any case, Insure++ proceeds as though the variable *definition* is correct, overriding the `extern` reference.

## Problem

In the following example, the file `baddecl1.c` declares the variable `a` to be a pointer,

```
1:      /*
2:      * File: baddecl1.c
3:      */
4:      int *a;
```

while the file `baddecl2.c` declares it to be an array type.

```
1:      /*
2:      * File: baddecl2.c
3:      */
4:      extern int a[];
5:
6:      main()
7:      {
8:          a[0] = 10;
9:          return (0);
10:     }
```

## Diagnosis (at runtime)

```
[baddecl2.c:4] **BAD_DECL**
1 >>          extern int a[];

2      Incompatible global declarations: a
```

```

3   Array and non-array declarations are not equivalent.
   Actual declaration:
       non-array (4 bytes), declared at baddecl1.c, 4
4   Conflicting declaration:
       array of unspecified size,
       declared at baddecl2.c, 4

```

1. Source line at which the problem was detected.
2. Description of the problem and the object whose declarations conflict.
3. Brief description of the conflict.
4. Information about the conflicting definitions, including the sizes of the declared objects and the locations of their declarations.

## Repair

The lines on which the conflicting declarations are made are both shown in the diagnostic report. They should be examined and the conflict resolved.

In the case shown here, for example, a suitable correction would be to change the declaration file to declare an array with a fixed size, e.g.,

```
baddecl1.c, 4:           int a[10];
```

An alternative correction would be to change the definition in `baddecl2.c` to indicate a pointer variable, e.g.,

```
baddecl2.c, 4:           extern int *a;
```

Note that this change on its own will not fix the problem. In fact, if you ran the program modified this way, you would get another error, `EXPR_NULL`, because the pointer `a` doesn't actually point to anything and is `NULL` by virtue of being a global variable, initialized to zero.

To make this version of the code correct, you would need to include something to allocate memory and store the pointer in `a`. For example,

```

1:      /*
2:      * File: baddecl2.c (modified)
3:      */
4:      #include <stdlib.h>
5:      extern int *a;

```

```
6:
7:     main()
8:     {
9:         a = (int *)malloc(10*sizeof(int));
10:        a[0] = 10;
11:    }
```

Some applications may genuinely need to declare objects with different sizes, in which case you can suppress error messages by suppressing `BAD_DECL` in the Suppressions Control Panel.

# BAD\_FORMAT

## Mismatch in format specification

This error is generated when a call to one of the `printf` or `scanf` routines contains a mismatch between a parameter type and the corresponding format specifier or the format string is nonsensical.

Insure++ distinguishes several types of mismatches which have different levels of severity as follows:

- `sign` - Types differ only by sign, e.g., `int` vs. `unsigned int`.
- `compatible` - Fundamental types are different but they happen to have the same representation on the particular hardware in use, e.g., `int` vs. `long` on machines where both are 32-bits, or `int *` vs. `long` where both are 32-bits.
- `incompatible` - Fundamental types are different, e.g. `int` vs. `double`.
- `other` - A problem other than an argument type mismatch is detected, such as passing the wrong number of arguments.

Error messages are classified according to this scheme and can be selectively enabled or disabled as described in the section “Repair” on page 217.

## Problem #1

An example of format type mismatch occurs when the format specifiers passed to one of the `printf` routines do not correspond to the data, as shown below.

```
1:      /*
2:      * File: badform1.c
3:      */
4:      main()
5:      {
6:          double f = 1.23;
7:          int i = 99;
8:
9:          printf("%d %f\n", f, i);
10:     }
```

This type of mismatch is detected during compilation.

### Diagnosis (during compilation)

```
1 [badform1.c:9] **BAD_FORMAT(incompatible)**
2     Wrong type passed to printf (argument 2).
   Expected int, found double.
>>     printf("%d %f\n", f, i);

[badform1.c:9] **BAD_FORMAT(incompatible)**
   Wrong type passed to printf (argument 3).
   Expected double, found int.
>>     printf("%d %f\n", f, i);
```

1. Source lines at which problems were detected.
2. Description of the problem and the arguments that are incorrect.

## Problem #2

A more dangerous problem occurs when the types passed as arguments to one of the `scanf` functions are incorrect. In the following code, for example, the call to `scanf` tries to read a double precision value, indicated by the `%lf` format, into a single precision value. This will overwrite memory.

```
1:     /*
2:     * File: badform2.c
3:     */
4:     main()
5:     {
6:         int a;
7:         float f;
8:
9:         scanf("%lf", &f);
10:    }
```

This problem is again diagnosed at compile time (along with the `WRITE_OVERFLOW`, which is not shown below).

### Diagnosis (during compilation)

```
1 [badform2.c:9] **BAD_FORMAT(incompatible)**
```

```

2      Wrong type passed to scanf (argument 2).
      Expected double *, found float *.
>>      scanf("%lf\n", &f);

```

1. Source lines at which problems were detected.
2. Description of the problem and the arguments that are incorrect.

### Problem #3

A third type of problem is caused when the format string being used is a variable rather than an explicit string. The following code contains an error handler that attempts to print out a message containing a filename and line number. In line 18 of the calling routine, however, the arguments are reversed.

```

1:      /*
2:      * File: badform3.c
3:      */
4:      char *file;
5:      int line;
6:
7:      error(format)
8:          char *format;
9:      {
10:         printf(format, file, line);
11:     }
12:
13:     main()
14:     {
15:         file = "foo.c";
16:         line = 3;
17:
18:         error("Line %d, file %s\n");
19:     }

```

### Diagnosis (at runtime)

```

[badform3.c:10] **BAD_FORMAT(incompatible)**
1 >>      printf(format, file, line);

2      Format string is inconsistent:
          Wrong type passed to printf (argument 3).
          Expected pointer, found int.
3      Format string: "Line %d, file %s\n"

```

```
Stack trace where the error occurred:  
4      error() badform3.c, 10  
      main() badform3.c, 18
```

1. Source line at which the problem was detected.
2. Description of the problem and the argument that is in error.
3. Explanation of the error and the format string that caused it.
4. Stack trace showing the function call sequence leading to the error.

The error diagnosed in this message is in the `incompatible` category, because any attempt to print a string by passing an integer variable will result in garbage. Note that with some compilers, this program may cause a core dump because of this error, while others will merely produce incorrect output.

There is, however, a second potential error in this code in the same line.

Because the arguments are in the wrong order in line 7, an attempt will be made to print a pointer variable as an integer. This error is in the `compatible` class, since a pointer and an integer are both the same size in memory. Since `compatible` `BAD_FORMAT` errors are suppressed by default, you will not see it. (These errors are suppressed because they tend to cause unexpected rather than incorrect behavior.)

If you enabled these errors, you would see a second problem report from this code.

Note: If you run `Insure++` on an architecture where pointers and integers are not the same length, then this second error would also be in the `incompatible` class and would be displayed by default.

## Repair

Most of these problems are simple to correct based on the information given. Normally, the correction is one or more of the following

- Change the format specifier used in the format string.
- Change the type of the variable involved.
- Add a suitable typecast.

For example, problem #1 can be corrected by simply changing the incorrect line of code as follows

```
badform1.c, line 9: printf("%d %f\n", i, f);
```

The other problems can be similarly corrected.

If your application generates error messages that you wish to ignore, you can suppress `BAD_FORMAT` in the Suppressions Control Panel.

This directive suppresses all `BAD_FORMAT` messages. If you wish to be more selective and suppress only a certain type of error, you can use the syntax

```
BAD_FORMAT(class1, class2, ...)
```

where the arguments are one or more of the identifiers for the various categories of errors described in “Mismatch in format specification” on page 214.

Similarly, you can enable suppressed types by unsuppressing them in the Suppressions Control Panel. The problem with the pointer and integer that was not shown in the current example could be displayed by unsuppressing `BAD_FORMAT(compatible)` in the Suppressions Control Panel. For an example of this option, as well as the remaining subcategories of `BAD_FORMAT`, see the example `badform4.c`.

# BAD\_INTERFACE

## Actual declaration of xxx conflicts with interface, or ignoring interface for xxx: conflicts with static or in-line declaration

This error will be generated any time there is a significant discrepancy between the source code being processed and an interface to one of the functions in the code. Common sources of this problem are redeclarations of standard system functions in your code.

### Problem

The following code shows a redeclaration of the function `printf` which will conflict with the version of the function expected by the interface.

```

1:      /*
2:      * File: badint.c
3:      */
4:      #include <stdio.h>
5:
6:      static void printf(i)
7:          int i;
8:      {
9:          fprintf(stdout, "%d\n", i);
10:     }
```

### Diagnosis (during compilation)

```

1 [badint.c:6] **BAD_INTERFACE**
2      Ignoring interface for printf: conflicts with static
      or inline declaration.
>>     static void printf(i)
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.

### Repair

There are several ways to approach solving this problem. The correct solution for your situation depends upon why the function was redefined in your code. If this is a version of the function that is used with all of your code, a permanent solution would be to write a new interface corresponding to your version of the function. A quicker, more temporary solution, appropriate if you only use this version of the function occasionally, would be to temporarily disable the checking of this interface using the `interface_ignore` Advanced Option. This option can be turned on and off on a per file basis as you work with different code which uses different versions of the function in question.

# BAD\_PARM

## Mismatch in argument type

This error is generated when an argument to a function or subroutine does not match the type specified in an earlier declaration or an interface file.

Insure++ distinguishes several types of mismatch which have different levels of severity as follows:

- `sign`: Types differ only by sign, e.g., `int` vs. `unsigned int`.
- `compatible`: Fundamental types are different but they happen to have the same representation on the particular hardware in use, e.g., `int` vs. `long` on machines where both are 32-bits.
- `incompatible`: Fundamental types are different, e.g. `int` vs. `float`.
- `union`: Forces a declared union argument to match only a similar union as an actual argument. If this is suppressed, you may pass any of the individual union elements to the routine, rather than the union type, or pass a union to a routine which expects one of the union-elements as an argument.
- `other`: An error was detected that is not simply a mismatched argument type, such as passing the wrong number of arguments to a function.
- `pointer`: This is not an error class, but a keyword used to suppress messages about mismatched pointer types, such as `int *` vs. `char *`. See “Repair” on page 225.

Error messages are classified according to this scheme and can be selectively enabled or disabled as described in the section “Repair” on page 225.

## Problem #1

The following shows an error in which an incorrect argument is passed to the function `foo`.

```

1:      /*
2:      * File: badparam1.c
3:      */
4:      void foo(str)
5:          char *str;
6:      {
7:          return;
8:      }
9:
10:     main()
11:     {
12:         int *iptr;
13:
14:         foo(iptr);
15:         return (0);
16:     }

```

This type of mismatch is detected during compilation.

### Diagnosis (during compilation)

```

1 [badparam1.c:14] **BAD_PARM(incompatible)**
2      Wrong type passed to foo (argument 1: str)
      Expected char *, found int *.
>>      foo(iptr)

```

1. Source lines at which problems were detected.
2. Description of the problem and the arguments that are incorrect.

### Problem #2

Another simple problem occurs when arguments are passed to functions in the wrong order, as in the following example.

```

1:      /*
2:      * File: badparam2.c
3:      */
4:      long foo(f, l)
5:          double f;
6:          long l;
7:      {
8:          return f+l;
9:      }
10:
11:     main()

```

```

12:     {
13:         long ret = foo(32L, 32.0);
14:
15:         printf("%ld\n", ret);
16:         return 0;
17:     }

```

## Diagnosis (during compilation)

```

1 [badparm2.c:13] **BAD_PARM(incompatible)**2
2     Wrong type passed to foo (argument 1: f)
   Expected double, found long.
>>         long ret = foo(32L, 32.0);

[badparm2.c:13] **BAD_PARM(incompatible)**
   Wrong type passed to foo (argument 2: l).
   Expected long, found double.
>>         long ret = foo(32L, 32.0);

```

1. Source lines at which problems were detected.
2. Description of the problem and the arguments that are incorrect.

## Problem #3

The following example illustrates the `BAD_PARM(union)` error category. The functions `func1` and `func2` expect to be passed a union and a pointer to an integer, respectively. The code in the `main` routine then invokes the two functions both properly and by passing the incorrect types.

Note that this code will probably work on most systems due to the internal alignment of the various data types. Relying on this behavior is, however, non-portable.

```

1:     /*
2:     * File: badparm3.c
3:     */
4:     union data {
5:         int i;
6:         double d;
7:     };
8:

```

```

9:      void func1(ptr)
10:         union data *ptr;
11:     {
12:         ptr->i = 1;
13:     }
14:
15:     void func2(p)
16:         int *p;
17:     {
18:         *p = 1;
19:     }
20:
21:     main()
22:     {
23:         int t;
24:         union data u;
25:
26:         func1(&u);
27:         func1(&t);           /* BAD_PARM */
28:         func2(&u);           /* BAD_PARM */
29:         func2(&t);
30:     }

```

## Diagnosis (during compilation)

```

1 [badparm3.c:27] **BAD_PARM(union)**
2      Wrong type passed to func1 (argument 1: ptr)
      Expected union data *, found int *.
>>          func1(&t);           /* BAD_PARM */

[badparm3.c:28] **BAD_PARM(union)**
      Wrong type passed to func2 (argument 1: p)
      Expected int *, found union data *.
>>          func2(&u);           /* BAD_PARM */

```

1. Source lines at which problems were detected.
2. Description of the problem and the arguments that are incorrect.

## Repair

Most of these problems are simple to correct based on the information given. For example, problem #1 can be corrected by simply changing the incorrect line of code as follows:

```
badparm1.c, line 6: if(strchr("testing", 's'))
```

The other problems can be similarly corrected.

If your application generates error messages that you wish to ignore, you can suppress `BAD_PARM` in the Suppressions Control Panel.

This directive suppresses all `BAD_PARM` messages. If you wish to be more selective and suppress only a certain type of error, you can use the syntax

```
BAD_PARM(class1, class2, ...)
```

where the arguments are one or more of the identifiers for the various categories of error described on “Mismatch in argument type” on page 221. Similarly, you can enable suppressed error messages by selecting **Unsuppress** in the **Action** field.

Thus, you could enable warnings about conflicts between types `int` and `long` (on systems where they are the same number of bytes) by unsuppressing

```
BAD_PARM(compatible)
```

In addition to the keywords described on “Mismatch in argument type” on page 221, you can also use the type `pointer` to suppress all messages about different pointer types.

For example, many programs declare functions with the argument type `char *`, which are then called with pointers to various other data types. The ANSI standard recommends that you use type `void *` in such circumstances, since this is allowed to match any pointer type. If, for some reason, you cannot do this, you can suppress messages from `Insure++` about incompatible pointer types by suppressing

```
BAD_PARM(pointer)
```

in the Suppressions Control Panel.

# COPY\_BAD\_RANGE

## Copying pointer which is out-of-range

This error is generated whenever an attempt is made to copy a pointer which points outside a valid range. It is not necessarily a serious problem, but may indicate faulty logic in the coding. Therefore, this error is suppressed by default.

### Problem

The following code illustrates the problem in a simple way. In line 7, the pointer `a` is initialized as an array of 10 `char`s. The next line then attempts to make pointer `b` point to an area which has not been allocated. The resulting pointer is not valid.

```

1:      /*
2:      * File: copybad.cpp
3:      */
4:      int main() {
5:          char *a, *b;
6:
7:          a = new char [10];
8:          b = a + 20;
9:          return 0;
10:     }
```

### Diagnosis (at runtime)

```

1 [copybad.cpp:8] **COPY_BAD_RANGE**
>>          b = a + 20;

2      Copying pointer which is out-of-range: a + 20

3      Pointer          : 0x0007c124
      Actual block    : 0x0007c110 thru 0x0007c119 (10 bytes)
      a, allocated at:
                          main()      copybad.cpp, 7

4      Stack trace where the error occurred:
                          main()      copybad.cpp, 8
```

1. Source line at which the problem was detected.
2. Brief description of the problem.
3. Description of the pointer which is out-of-range.
4. Stack trace showing the function call sequence leading to the error.

## Repair

The simple way to avoid this problem is to not copy the invalid pointer. There may be an incorrect boundary case in your code causing the problem.

# COPY\_DANGLING

## Copying pointer which has already been freed

This error is generated whenever an attempt is made to copy a pointer to a block of memory which has been freed. This error is suppressed by default.

### Problem

The following code illustrates the problem in a simple way. In line 7, the pointer `a` is freed by calling `delete[]`. The next line then attempts to copy from the address `a` into the variable `b`. Since `a` has already been freed, `b` will not point to valid memory either.

```

1:      /*
2:      * File: copydang.cpp
3:      */
4:      int main() {
5:          char *a = new char [10], *b;
6:
7:          delete[] a;
8:          b = a;
9:          return 0;
10:     }
```

### Diagnosis (at runtime)

```

1 [copydang.cpp:8] **COPY_DANGLING**
  >>          b = a;

2          Copying dangling pointer: a

3          Pointer   : 0x0007b6a0
          In block  : 0x0007ebc0 thru 0x0007ebc9 (10 bytes)
                   a, allocated at:
                           main()    copydang.cpp, 5

4          stack trace where memory was freed:
                           main()    copydang.cpp, 7

5          Stack trace where the error occurred:
                           main()    copydang.cpp, 8
```

1. Source line at which the problem was detected.
2. Brief description of the problem.
3. Description of the pointer which is dangling.
4. Stack trace showing where the dangling pointer was freed.
5. Stack trace showing the function call sequence leading to the error.

## Repair

The simple way to avoid this problem is to not attempt to reuse pointers after they have been freed. Check that the deallocation that occurs at the indicated location should have taken place. Also check if pointer you are (mis)using should be pointing to a block allocated at the indicated place.

# COPY\_UNINIT\_PTR

## Copying uninitialized pointer

This error is generated whenever an uninitialized pointer is copied.

**Note:** This error category will be disabled if full uninitialized memory checking is in effect (the default). In this case, errors are detected in the `READ_UNINIT_MEM` category instead.

## Problem

The pointer `a` is declared in line 5, but is never initialized. Therefore, when an attempt is made in line 7 to copy this pointer to `b`, an error is generated.

```

1:      /*
2:      * File: copyunin.cpp
3:      */
4:      int main() {
5:          char *a, *b;
6:
7:          b = a;
8:          return 0;
9:      }
```

## Diagnosis (at runtime)

```

1 [copyunin.cpp:7] **COPY_UNINIT_PTR**
  >>          b = a;

2          Copying uninitialized pointer: a

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

1. Source line at which the problem was detected.
2. Brief description of the problem.
3. Stack trace showing the function call sequence leading to the error.

## Repair

This problem is usually caused by omitting an assignment or allocation statement that would initialize a pointer. The example code given could be corrected by assigning a value to `a` before reaching line 7.

# COPY\_WILD

## Copying wild pointer

This problem occurs when an attempt is made to copy a pointer whose value is invalid or which Insure++ did not see allocated.

This can come about in a couple of ways.

- Errors in user code that result in pointers that don't point at any known memory block.
- Compiling only *some* of the files that make up an application. This can result in Insure++ not knowing enough about memory usage to distinguish correct and erroneous behavior.

**Note:** This section focuses on the first type of problem described above. For information about the second type of problem, contact ParaSoft's Quality Consultants.

## Problem

The following code attempts to use the address of a variable but contains an error at line 9; the address operator (&) has been omitted.

```
1:      /*
2:      * File: copywild.c
3:      */
4:
5:      main()
6:      {
7:          int a = 123, *b;
8:
9:          b = a;
10:         return (0);
11:     }
```

## Diagnosis (at runtime)

```
[copywild.c:9] **COPY_WILD**
1 >>          b = a;

2          Copying wild pointer: a
```

```
3     Pointer : 0x0000007b
      Stack trace where the error occurred:
4     main() copywild.c, 9
```

1. Source line at which the problem was detected.
2. Description of the problem and the name of the parameter that is in error.
3. Value of the bad pointer.
4. Stack trace showing the function call sequence leading to the error.

Note that most compilers will generate warning messages for this error since the assignment uses incompatible types.

# DEAD\_CODE

## Code is not executed

This error is generated when code is not evaluated, has no effect, or is unreachable. Insure++ distinguishes between several types of dead code as follows:

- `emptystmt` - The statement is empty.
- `emptyloopbody` - Loop body is empty.
- `noeffect` - Code has no effect.
- `notevaluated` - Code is not evaluated.

Error messages are classified according to this scheme and can be selectively enabled or disabled. By default, this error category is suppressed.

## Problem #1

The following code shows a very tricky, well-disguised error that demonstrates how hard it is to find problems of this type without Insure++. The initialization function `get_glob` is never called by this code. Because `func_X2` is declared as a static function in the `Global` class, it can be called directly by `main`. This is in fact what happens, meaning that line 23 is interpreted as only a call to `func_X2`. Therefore, an error is generated since the call to `get_glob` is never evaluated.

```
1:      /*
2:      * File: deadcode.cpp
3:      */
4:      #include <iostream.h>
5:
6:      class Global {
7:      public:
8:          int j;
9:          static int func_X2(int i);
10:     };
11:
12:     int Global::func_X2(int i) {
13:         return i*2;
14:     }
15:
```

```

16:     Global *get_glob() {
17:         cerr << "Initializing..."
18:     << endl;
19:         return (Global *) 0;
20:     }
21:
22:     int main() {
23:         get_glob()->func_X2(2);
24:         return 0;
25:     }

```

## Diagnosis (during compilation)

```

1 [deadcode.cpp:23] **DEAD_CODE(notevaluated)**
2     Code is not evaluated
  >>         get_glob()->func_X2(2);

```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is incorrect.

## Repair

This problem can be solved by replacing line 23 with a direct call to `func_X2`:

```
Global::func_X2(2);
```

or by not making `func_X2` a static function.

In some cases, it may be that the dead code was never intended to be called. If that is the case, the dead code should be eliminated for clarity.

## Problem #2

The following code illustrates several other types of `DEAD_CODE` errors, this time in C.

```

1:     /*
2:     * File: deadcode.c
3:     */
4:     int main()
5:     {
6:         int i = 0;
7:
8:         ;

```

```

9:             i;
10:            for (i; i; i)
11:                ;
12:            return 0;
13:        }

```

### Diagnosis (during compilation)

```

1 [deadcode.c:8] **DEAD_CODE(emptystmt)**
2     Statement is empty
>>         ;
[deadcode.c:9] **DEAD_CODE(noeffect)**
3     Code has no effect
>>         i;
[deadcode.c:10] **DEAD_CODE(noeffect)**
4     For loop initializer has no effect
>>         for (i; i; i)
[deadcode.c:10] **DEAD_CODE(noeffect)**
5     For loop increment has no effect
>>         for (i; i; i)
[deadcode.c:11] **DEAD_CODE(emptyloopbody)**
6     Loop body is empty (may be okay)
>>         ;

```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is incorrect.

### Repair

These errors are usually corrected by removing the superfluous statement or by modifying the statement so that it does what it was intended to do, e.g., add a missing increment operator. An empty loop body may be useful in certain situations. In such a case, you may want to suppress that subcategory of `DEAD_CODE`.

# DELETE\_MISMATCH

## 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` may 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.

### Problem #1

The following code 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:      }
```

### Diagnosis (at runtime)

```

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

2          Inconsistent usage of delete operator: a

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

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

1. Source line at which the problem was detected.
2. Description of the problem and the operator which doesn't match.
3. Brief description of the mismatch.
4. Stack trace showing the function call sequence leading to the error.

## Problem #2

The following code shows a block of memory allocated with `new`, without `[]`, and freed with `delete[]`. This may cause some implementations of C++ to crash, because the compiler may look for extra bits of information about how the block was allocated. Some compilers allow this type of error, extending the ANSI standard. In this case, there would be no extra bits, so the compiler would attempt to read from an invalid memory address.

```

1:      /*
2:      * File: delmis2.cpp
3:      */
4:
5:      int main() {
6:          int *a = new int;
7:          delete[] a;
8:          return 0;
9:      }
```

## Diagnosis (at runtime)

```

1 [delmis2.cpp:7] **DELETE_MISMATCH**
  >>          delete[] a;

2          Inconsistent usage of delete operator: a

3          [] used to delete a non-array
              a, allocated at:
                  main() delmis2.cpp, 6

4          Stack trace where the error occurred:
                  main()    delmis2.cpp, 7
```

1. Source line at which the problem was detected.
2. Description of the problem and the operator which doesn't match.

3. Brief description of the mismatch.
4. Stack trace showing the function call sequence leading to the error.

## Repair

To eliminate this error, you need to change the `delete` call to match the `new` call. In our first example, this could be accomplished by calling `delete[]` instead of `delete`, and vice versa in the second example.

# EXPR\_BAD\_RANGE

## Expression exceeded range

This error is generated whenever an expression uses a pointer that is outside its legal range. In many circumstances, these pointers are then turned into legal values before use (code generated by automated programming tools such as `lex` and `yacc`), so this error category is suppressed by default. If used with their illegal values, other Insure++ errors will be displayed which can be tracked to their source by re-enabling this error class.

## Problem

In this code, the pointer `a` initially points to a character string. It is subsequently incremented beyond the end of the string. When the resulting pointer is used to make an array reference, a range error is generated.

```

1:      /*
2:      * File: exprange.c
3:      */
4:      main()
5:      {
6:          char *a = "test";
7:          char *b;
8:
9:          a += 6;
10:         b = &a[1];
11:         return (0);
12:     }
```

## Diagnosis (at runtime)

```

1 [exprange.c:10] **EXPR_BAD_RANGE**
  >>          b = &a[1];

2          Expression exceeded range: a[1]

          Index used: 1
3          Pointer          : 0x0000e226
          In block         : 0x0000e220 thru 0x0000e224 (5 bytes)
                           a, declared at exprange.c, 6
```

```
4      Stack trace where the error occurred:  
      main() exprange.c, 10
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Description of the memory block to which the out of range pointer used to point, including the location at which it is declared.
4. Stack trace showing the function call sequence leading to the error.

## Repair

In most cases, this error is caused by incorrect logic in the code immediately prior to that at which the message is generated. Probably the simplest method of solution is to run the program under a debugger with a breakpoint at the indicated location.

If you cannot find the error by examining the values of other variables at this location, the program should be run again, stopped somewhere shortly before the indicated line, and single-stepped until the problem occurs.

# EXPR\_DANGLING

## Expression uses dangling pointer

This error is generated whenever an expression operates on a dangling pointer - i.e., one which points to either

- A block of dynamically allocated memory that has already been freed.
- A block of memory which was allocated on the stack in some routine that has subsequently returned.

## Problem

The following code fragment shows a block of memory being allocated and then freed. After the memory is de-allocated, the pointer to it is used again, even though it no longer points to valid memory.

```

1:      /*
2:      * File: expdangl.c
3:      */
4:      #include <stdlib.h>
5:
6:      main()
7:      {
8:          char *a = (char *)malloc(10);
9:          char b[10];
10:
11:         free(a);
12:         if(a > b)
13:             a = b;
14:         return (0);
15:     }

```

## Diagnosis (at runtime)

```

[expdangl.c:12] **EXPR_DANGLING**
1 >>         if(a > b)

2     Expression uses dangling pointer: a > b

3     Pointer      : 0x00013868

```

```
In block      : 0x00013868 thru 0x00013871 (10 bytes)
                block allocated at:
                malloc() (interface)
                main() expdangl.c, 8
                stack trace where memory was freed:
                main() expdangl.c, 11

4  Stack trace where the error occurred:
    main() expdangl.c, 12
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Description of the memory block to which the pointer used to point, including the location at which it was allocated and subsequently freed.
4. Stack trace showing the function call sequence leading to the error.

## Repair

A good first check is to see if the pointer used in the expression at the indicated line is actually the one intended.

If it appears to be the correct pointer, check the line of code where the block was freed (as shown in the error message) to see if it was freed incorrectly.

# EXPR\_NULL

## Expression uses NULL pointer

This error is generated whenever an expression operates on the `NULL` pointer.

### Problem

The following code fragment declares a pointer, `a`, which is initialized to zero by virtue of being a global variable. It then manipulates this pointer, generating the `EXPR_NULL` error.

```

1:      /*
2:      * File: expnull.c
3:      */
4:      char *a;
5:
6:      main()
7:      {
8:          char *b;
9:
10:         b = &a[1];
11:         return (0);
12:     }
```

### Diagnosis (at runtime)

```

1 [expnull.c:10] **EXPR_NULL**
  >>          b = &a[1];

2          Expression uses null pointer: a[1]
3          Stack trace where the error occurred:
           main() expnull.c, 10
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Stack trace showing the function call sequence leading to the error.

## Repair

One potential cause of this error is shown in this example. The pointer `a` is a global variable that will be initialized to zero by the compiler. Since this variable is never modified to point to anything else, it is still `NULL` when first used.

One way the given code can be corrected is by adding an assignment as follows

```
/*
 * File: expnull.c (modified)
 */
char *a;
main()
{
    char *b, c[10];
    a = c;
    b = &a[1];
    return (0);
}
```

It can also be corrected by allocating a block of memory.

A second possibility is that the pointer was set to zero by the program at some point before its subsequent use and not re-initialized. This is common in programs which make heavy use of dynamically allocated memory and which mark freed blocks by resetting their pointers to `NULL`.

A final common problem is caused when one of the dynamic memory allocation routines, `malloc`, `calloc`, or `realloc`, fails and returns a `NULL` pointer. This can happen either because your program passes bad arguments or simply because it asks for too much memory. A simple way of finding this problem with `Insure++` is to enable the `RETURN_FAILURE` error code (see “`RETURN_FAILURE`” on page 324) with your Advanced Options and run the program again. It will then issue diagnostic messages every time a system call fails, including the memory allocation routines.

# EXPR\_UNINIT\_PTR

## Expression uses uninitialized pointer

This error is generated whenever an expression operates on an uninitialized pointer.

### Problem

The following code uses an uninitialized pointer.

```

1:      /*
2:      * File: expuptr.c
3:      */
4:      main()
5:      {
6:          char *a, b[10], c[10];
7:
8:          if (a > b)
9:              a = b;
10:         return (0);
11:     }
```

### Diagnosis (at runtime)

```

1 [expuptr.c:8] **EXPR_UNINIT_PTR**
  >>          if (a > b)

2          Expression uses uninitialized pointer: a > b
3          Stack trace where the error occurred:
          main() expuptr.c, 8
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Stack trace showing the function call sequence leading to the error.

## Repair

This error is normally caused by omitting an assignment statement for the uninitialized variable. The example code can be corrected as follows:

```
1:      /*
2:      * File: expuptr.c (modified)
3:      */
4:      main()
5:      {
6:          char *a, b[10], c[10];
7:
8:          a = c;
9:          if (a > b)
10:             a = b;
11:          return (0);
12:     }
```

## EXPR\_UNRELATED\_PTRCMP

### Expression compares unrelated pointers

This error is generated whenever an expression tries to compare pointers that do not point into the same memory block. This only applies to the operators `>`, `>=`, `<`, and `<=`. The operators `==` and `!=` are exempt from this case.

The ANSI C-language specification declares this construct undefined except in the special case where a pointer points to an address one past the end of a block.

### Problem

The following code illustrates the problem by comparing pointers to two data objects.

```
1:          /*
2:          * File: expucmp.c
3:          */
4:          #include <stdlib.h>
5:
6:          main()
7:          {
8:              char a[10], *b;
9:
10:             b = (char *)malloc(10);
11:
12:             if(a > b) a[0] = 'x';
13:             else a[0] = 'y';
14:             return (0);
15:          }
```

Note that the error in this code is not that the two objects `a` and `b` are of different data types (array vs. dynamic memory block), but that the comparison in line 12 attempts to compare pointers which do not point into the same memory block. According to the ANSI specification, this is an undefined operation.

## Diagnosis (at runtime)

```

1 [expucmp.c:12] **EXPR_UNRELATED_PTRCMP**
  >>          if(a > b) a[0] = 'x';

2      Expression compares unrelated pointers: a > b

      Left hand side: 0xf7fffb8c
3      In block: 0xf7fffb8c thru 0xf7fffb95 (10 bytes)
          a, declared at expucmp.c, 8

      Right hand side: 0x00013870
      In block: 0x00013870 thru 0x00013879 (10 bytes)
          block allocated at:
          malloc() (interface)
          main() expucmp.c, 10
4      Stack trace where the error occurred:
          main() expucmp.c, 12

```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Description of the two pointers involved in the comparison. For each pointer, the associated block of memory is shown together with its size and the line number at which it was declared or allocated.
4. Stack trace showing the function call sequence leading to the error.

## Repair

While this construct is technically undefined according to the ANSI C specification, it is supported on many machines and its use is fairly common practice. If your application genuinely needs to use this construct, you can suppress this message by suppressing

```
EXPR_UNRELATED_PTRCMP
```

in the Suppressions Control Panel.

# EXPR\_UNRELATED\_PTRDIFF

## Expression subtracts unrelated pointers

This error is generated whenever an expression tries to compute the difference between pointers that do not point into the same memory block.

The ANSI C language specification declares this construct undefined except in the special case where a pointer points to an object one past the end of a block.

## Problem

The following code illustrates the problem by subtracting two pointers to different data objects.

```

1:      /*
2:      * File: expudiff.c
3:      */
4:      #include <stdlib.h>
5:
6:      main()
7:      {
8:          char a[10], *b;
9:          int d;
10:
11:          b = (char *)malloc(10);
12:          d = b - a;
13:          return (0);
14:      }

```

## Diagnosis (at runtime)

```

[expudiff.c:12] **EXPR_UNRELATED_PTRDIFF**
1 >>          d = b - a;

2          Expression subtracts unrelated pointers: b - a

          Left hand side      : 0x00013878
3          In block   : 0x00013878 thru 0x00013881 (10 bytes)
          b, allocated at:
          malloc() (interface)

```

```

main() expudiff.c, 11

Right hand side      : 0xf7fffb8c
In block   : 0xf7fffb8c thru 0xf7fffb95 (10 bytes)
              a, declared at expudiff.c, 8
4 Stack trace where the error occurred:
      main() expudiff.c, 12

```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Description of the two pointers involved in the expression. For each pointer the associated block of memory is shown together with its size and the line number at which it was declared or allocated.
4. Stack trace showing the function call sequence leading to the error.

## Repair

While this construct is undefined according to the ANSI C language specification, it is supported on many machines and its use is fairly common practice. If your application genuinely needs to use this construct, you can suppress error messages by suppressing

```
EXPR_UNRELATED_PTRDIFF
```

in the Suppressions Control Panel.

# EXPR\_WILD

## Expression uses wild pointer

This error is generated whenever a program operates on a memory region that is unknown to Insure++. This can come about in two ways:

- Errors in user code that result in pointers that don't point at any known memory block.
- Compiling only *some* of the files that make up an application. This can result in Insure++ not knowing enough about memory usage to distinguish correct and erroneous behavior.

**Note:** This section focuses on the first type of problem described here. For information about the second type of problem, contact ParaSoft's Quality Consultants.

## Problem #1

The following code attempts to use the address of a local variable but contains an error at line 8 - the address operator (&) has been omitted.

```

1:      /*
2:      * File: expwld1.c
3:      */
4:      main()
5:      {
6:          int i = 123, j=345, *a;
7:
8:          a = i;
9:          if(a > &i)
10:             a = &j;
11:          return (0);
12:     }
```

## Diagnosis (at runtime)

```

[expwld1.c:9] **EXPR_WILD**
1 >>          if(a > &i)

2          Expression uses wild pointer: a > &i
```

```

3      Pointer : 0x0000007b
4      Stack trace where the error occurred:
          main() expwld1.c,

```

1. Source line at which the problem was detected.
2. Description of the problem and the name of the parameter that is in error.
3. Value of the wild pointer.
4. Stack trace showing the function call sequence leading to the error.

Keep in mind that most compilers will generate warning messages for this error since the assignment in line 8 uses incompatible types.

## Problem #2

A more insidious version of the same problem can occur when using `union` types. The following code first assigns the pointer element of a union but then overwrites it with another element before finally attempting to use it.

```

1:      /*
2:      * File: expwld2.c
3:      */
4:      union {
5:          int *ptr;
6:          int ival;
7:      } u;
8:
9:      main()
10:     {
11:         int i = 123, j=345;
12:
13:         u.ptr = &i;
14:         u.ival = i;
15:         if(u.ptr > &j)
16:             u.ptr = &j;
17:         return (0);
18:     }

```

Note that this code will not generate compile time errors.

## Diagnosis (at runtime)

```
[expwld2.c:15] **EXPR_WILD**  
1 >>          if(u.ptr > &j)  
  
2          Expression uses wild pointer: u.ptr > &j  
  
3          Pointer : 0x0000007b  
  
          Stack trace where the error occurred:  
4          main() expwld2.c, 15
```

1. Source line at which the problem was detected.
2. Description of the problem and the name of the parameter that is in error.
3. Value of the bad pointer.
4. Stack trace showing the function call sequence leading to the error.

## Repair

The simpler types of problem are most conveniently tracked in a debugger by stopping the program at the indicated source line. You should then examine the illegal value and attempt to see where it was generated. Alternatively you can stop the program at some point prior to the error and single-step it through the code leading up to the error.

“Wild pointers” can also be generated when Insure++ has only partial information about your program’s structure. For more information on this topic, contact ParaSoft’s Quality Consultants.

# FREE\_BODY

## Freeing memory block from body

This error is generated when an attempt is made to de-allocate memory by using a pointer which currently points into the middle of a block, rather than to its beginning.

### Problem

The following code attempts to free a memory region using an invalid pointer.

```

1:      /*
2:      * File: freebody.c
3:      */
4:      #include <stdlib.h>
5:
6:      main()
7:      {
8:          char *a = (char *)malloc(10);
9:          free(a+1);
10:     }
```

### Diagnosis (at runtime)

```

[freebody.c:9] **FREE_BODY**
1 >>          free(a+1);

2          Freeing memory block from body: a + 1

3          Pointer      : 0x000173e9
          Stack trace where the error occurred:
4              main() freebody.c, 9

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

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Value of the pointer that is being deallocated.

FREE\_BODY

4. Stack trace showing the function call sequence leading to the error.
5. Informational message indicating that a serious error has occurred which may cause the program to crash.

## Repair

This is normally a serious error. In most cases, the line number indicated in the diagnostics will have a simple error that can be corrected.

# FREE\_DANGLING

## Freeing dangling pointer

This error is generated when a memory block is freed multiple times.

### Problem

The following code frees the same pointer twice.

```

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

```

### Diagnosis (at runtime)

```

[freedngl.c:10] **FREE_DANGLING**
1 >>         free(a);

2           Freeing dangling pointer: a

3           Pointer   : 0x000173e0
           In block  : 0x000173e0 thru 0x000173e9 (10 bytes)
                   block allocated at:
4                               malloc() (interface)
                               main() freedngl.c, 8

5           stack trace where memory was freed:
                   main() freedngl.c, 9

6           Stack trace where the error occurred:
                   main() freedngl.c, 10

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

```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Value of the pointer that is being deallocated.
4. Information about the block of memory addressed by this pointer, including information about where this block was allocated.
5. Stack trace showing where this block was freed.
6. Stack trace showing the function call sequence leading to the error.
7. Informational message indicating that a serious error has occurred which may cause the program to crash.

## Repair

Some systems allow memory blocks to be freed multiple times. However, this is not portable and is not a recommended practice.

The information supplied in the diagnostics will allow you to see the line of code which previously de-allocated this block of memory. You should attempt to remove one of the two calls.

If your application is unable to prevent multiple calls to deallocate the same block, you can suppress error messages by suppressing

`FREE_DANGLING`

in the Suppressions Control Panel.

# FREE\_GLOBAL

## Freeing global memory

This error is generated if the address of a global variable is passed to a routine that de-allocates memory.

## Problem

The following code attempts to deallocate a global variable that was not dynamically allocated.

```

1:      /*
2:      * File: freeglob.c
3:      */
4:      char a[10];
5:
6:      main()
7:      {
8:          free(a);
9:          return (0);
10:     }
```

## Diagnosis (at runtime)

```

[freeglob.c:8] **FREE_GLOBAL**
1 >>          free(a);

2          Freeing global memory: a

3          Pointer   : 0x00012210
          In block  : 0x00012210 thru 0x00012217 (8 bytes)
4                      a,declared at freeglob.c, 4

          Stack trace where the error occurred:
5                      main() freeglob.c, 8

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

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.

3. Value of the pointer that is being deallocated.
4. Information about the block of memory addressed by this pointer, including information about where this block was declared.
5. Stack trace showing the function call sequence leading to the error.
6. Informational message indicating that a serious error has occurred which may cause the program to crash.

## Repair

Some systems allow this operation, since they keep track of which blocks of memory are actually dynamically allocated, but this is not portable programming practice and is not recommended.

In some cases, this error will result from a simple coding mistake at the indicated source line which can be quickly corrected.

A more complex problem may arise when a program uses both statically and dynamically allocated blocks in the same way. A common example is a linked list in which the head of the list is static, while the other entries are allocated dynamically. In this case, you must take care not to free the static list head when removing entries.

If your application is unable to distinguish between global and dynamically allocated memory blocks, you can suppress error messages by suppressing

`FREE_GLOBAL`

in the Suppressions Control Panel.

# FREE\_LOCAL

## Freeing local memory

This error is generated if the address of a local variable is passed to `free`.

### Problem

The following code attempts to free a local variable that was not dynamically allocated.

```

1:      /*
2:      * File: freelocl.c
3:      */
4:      main()
5:      {
6:          char b, *a;
7:
8:          a = &b;
9:          free(a);
10:         return (0);
11:     }
```

### Diagnosis (at runtime)

```

[freelocl.c:9] **FREE_LOCAL**
1 >>          free(a);

2          Freeing local memory: a

3          Pointer           : 0xf7fffb0f
          In block          : 0xf7fffb0f thru 0xf7fffb0f (1 byte)
4                                     b,declared at freelocl.c, 6

          Stack trace where the error occurred:
5          main() freelocl.c, 9

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

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Value of the pointer that is being deallocated.

4. Information about the block of memory addressed by this pointer, including information about where this block was declared.
5. Stack trace showing the function call sequence leading to the error.
6. Informational message indicating that a serious error has occurred which may cause the program to crash.

## Repair

Some systems allow this operation since they keep track of which blocks of memory are actually dynamically allocated, but this is not portable programming practice and is not recommended.

In most cases, this error will result from a simple coding mistake at the indicated source line which can be quickly corrected.

If your application is unable to distinguish between local variables and dynamically allocated memory blocks, you can suppress error messages by suppressing

`FREE_LOCAL`

in the Suppressions Control Panel.

# FREE\_UNINIT\_PTR

## Freeing uninitialized pointer

This error is generated whenever an attempt is made to de-allocate memory by means of an uninitialized pointer.

### Problem

This code attempts to free a pointer which has not been initialized.

```

1:      /*
2:      * File: freeuptr.c
3:      */
4:      main()
5:      {
6:          char *a;
7:          free(a);
8:          return (0);
9:      }
```

### Diagnosis (at runtime)

```

[freeuptr.c:7] **FREE_UNINIT_PTR**
1 >>          free(a);

2          Freeing uninitialized pointer: a
          Stack trace where the error occurred:
3              main() freeuptr.c, 7

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

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Stack trace showing the function call sequence leading to the error.
4. Informational message indicating that a serious error has occurred which may cause the program to crash.

## Repair

Some systems appear to allow this operation, since they will refuse to free memory that was not dynamically allocated. Relying on this behavior is very dangerous, however, since an uninitialized pointer may “accidentally” point to a block of memory that *was* dynamically allocated, but should not be freed.

# FREE\_WILD

## Freeing wild pointer

This error is generated when memory is de-allocated that is unknown to Insure++. This can come about in two ways:

- Errors in user code that result in pointers that don't point at any known memory block.
- Compiling only *some* of the files that make up an application. This can result in Insure++ not knowing enough about memory usage to distinguish correct and erroneous behavior.

**Note:** This section focuses on the first type of problem described here. For information on the second type of problem, contact ParaSoft's Quality Consultants.

A particularly unpleasant problem can occur when using `union` types. The following code first assigns the pointer element of a union but then overwrites it with another element before finally attempting to free the initial memory block.

```
1:      /*
2:      * File: freewild.c
3:      */
4:      #include <stdlib.h>
5:
6:      union {
7:          int *ptr;
8:          int ival;
9:      } u;
10:
11:     main()
12:     {
13:         char *a = (char *)malloc(100);
14:
15:         u.ptr = a;
16:         u.ival = 123;
17:         free(u.ptr);
18:         return (0);
19:     }
```

## Diagnosis (at runtime)

```
[freewild.c:17] **FREE_WILD**  
1 >>                free(u.ptr);  
  
2           Freeing wild pointer: u.ptr  
  
3           Pointer : 0x0000007b  
  
           Stack trace where error occurred:  
4           main() freewild.c, 17
```

1. Source line at which the problem was detected.
2. Description of the problem and the name of the parameter that is in error.
3. Value of the bad pointer.
4. Stack trace showing the function call sequence leading to the error.

## Repair

This problem is most conveniently tracked in a debugger by stopping the program at the indicated source line. You should then examine the illegal value and attempt to see where it was generated. Alternatively you can stop the program at some point prior to the error and single-step through the code leading up to the problem.

“Wild pointers” can also be generated when Insure++ has only partial information about your program’s structure. Contact ParaSoft’s Quality Consultants for more information on this topic.

# FUNC\_BAD

## Function pointer is not a function

This error is generated when an attempt is made to call a function through either an invalid or unknown function pointer.

### Problem

One simple way to generate this error is through the use of the `union` data type. If the union contains a function pointer which is invoked after initializing some other union member, this error can occur.

```

1:  /*
2:   * File: funcbad.c
3:   */
4:  union {
5:      int *iptr;
6:      int (*fptr)();
7:  } u;
8:
9:  main()
10: {
11:     int i;
12:
13:     u.iptr = &i;
14:     u.fptr();
15:     return (0);
16: }
```

### Diagnosis (at runtime)

```

[funcbad.c:14] **FUNC_BAD**
1 >>                u.fptr();

2                Function pointer is not a function: u.fptr

3                Pointer                : 0xf7fff8cc
                In block                : 0xf7fff8cc thru 0xf7fff8cf
4                (4 bytes,1 element)
```

```
                                i, declared at funcbad.c, 11
Stack trace where the error occurred:
5      main() funcbad.c, 14
6
    **Memory corrupted. Program may crash!**
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. The value of the pointer through which the call is being attempted.
4. Description of the memory block to which this pointer actually points, including its size and the source line of its declaration.
5. Stack trace showing the function call sequence leading to the error.
6. Informational message indicating that a serious error has occurred which may cause the program to crash.

## Repair

The description of the memory block to which the pointer points should enable you to identify the statement which was used to assign the function pointer incorrectly.

# FUNC\_NULL

## Function pointer is NULL

This error is generated when a function call is made via a `NULL` function pointer.

### Problem

This code attempts to call a function through a pointer that has never been explicitly initialized. Since the pointer is a global variable, it is initialized to zero by default, resulting in the attempt to call a `NULL` pointer.

```

1:      /*
2:      * File: funcnull.c
3:      */
4:      void (*a)();
5:
6:      main()
7:      {
8:          a();
9:          return (0);
10:     }
```

### Diagnosis (at runtime)

```

[funcnull.c:8] **FUNC_NULL**
1 >>      a();

2      Function pointer is null: a
      Stack trace where the error occurred:
3          main() funcnull.c, 8

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

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Stack trace showing the function call sequence leading to the error.

4. Informational message indicating that a serious error has occurred which may cause the program to crash.

## Repair

The most common way to generate this problem is the one shown here, in which the pointer never was explicitly initialized and is set to zero. This case normally requires the addition of an assignment statement prior to the call as shown below.

```
/*
 * File: funcnull.c (modified)
 */
void (*a)();
extern void myfunc();

main()
{
    a = myfunc;
    a();
    return (0);
}
```

A second fairly common programming practice is to terminate arrays of function pointers with `NULL` entries. Code that scans a list looking for a particular function may end up calling the `NULL` pointer if its search criterion fails. This normally indicates that protective programming logic should be added to prevent against this case.

# FUNC\_UNINIT\_PTR

## Function pointer is uninitialized

This error is generated when a call is made through an uninitialized function pointer.

### Problem

This code attempts to call a function through a pointer that has not been set.

```

1:      /*
2:      * File: funcuptr.c
3:      */
4:      main()
5:      {
6:          void (*a)();
7:
8:          a();
9:          return (0);
10:     }
```

### Diagnosis (at runtime)

```

[funcuptr.c:8] **FUNC_UNINIT_PTR**
1 >>          a();

2          Function pointer is uninitialized: a
3          Stack trace where the error occurred:
           main() funcuptr.c, 8

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

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Stack trace showing the function call sequence leading to the error.
4. Informational message indicating that a serious error has occurred which may cause the program to crash.

## Repair

This problem normally occurs because some assignment statement has been omitted from the code. The current example can be fixed as follows:

```
extern void myfunc();

main()
{
    void (*a)();
    a = myfunc;
    a();
}
```

# INSURE\_ERROR

## Internal errors (various)

This error code is reserved for fatal errors that Insure++ is unable to deal with adequately such as running out of memory, or failing to open a required file.

Unrecognized string values in the Windows Registry or Advanced Options can also generate this error.

# INSURE\_WARNING

## Errors from `iic_warning` calls

This error code is generated when Insure++ encounters a call to the `iic_warning` interface function.

## Example

The following code contains a call to a function called `archaic_function` whose use is to be discouraged.

```

1:      /*
2:      * File: warn.c
3:      */
4:      #include <stdio.h>
5:
6:      main()
7:      {
8:          archaic_function();
9:          exit(0);
10:     }
```

In order to use the `iic_warning` capability, we can make an interface to the `archaic_function` as follows.

```

1:      /*
2:      * File: warn_i.c
3:      */
4:      void archaic_function(void)
5:      {
6:          iic_warning(
7:              "This function is obsolete");
8:          archaic_function();
9:      }
```

## Diagnosis (during compilation)

```

1 [warn.c:8] **INSURE_WARNING**
2          Use of archaic_function is deprecated.
>>          archaic_function();
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.

## Repair

This error category is suppressed by default, so you must unsuppress

`INSURE_WARNING`

in theSuppressions Control Panel before compiling code which uses it.

There are many uses for `iic_warning` and the `INSURE_WARNING` error, so no specific suggestions for error correction are appropriate. Hopefully, the messages displayed by the system will provide sufficient assistance.

# LEAK\_ASSIGN

## Memory leaked due to pointer reassignment

This error is generated whenever a pointer assignment occurs which will prevent a block of dynamically allocated memory from ever being freed. Normally this happens because the pointer being changed is the only one that still points to the dynamically allocated block.

### Problem

This code allocates a block of memory, but then reassigns the pointer to the block to a static memory block. As a result, the dynamically allocated block can no longer be freed.

```

1:      /*
2:      * File: leakasgn.c
3:      */
4:      #include <stdlib.h>
5:
6:      main()
7:      {
8:          char *b, a[10];
9:
10:         b = (char *)malloc(10);
11:         b = a;
12:         return (0);
13:     }

```

### Diagnosis (at runtime)

```

[leakasgn.c:11] **LEAK_ASSIGN**
1 >>         b = a;

2         Memory leaked due to pointer reassignment: <return>

3         Lost block:           0x000173e8 thru 0x000173f1 (10 bytes)
                                block allocated at:
                                    malloc() (interface)
                                    main() leakasgn.c, 10

Stack trace where the error occurred:

```

```
4 main() leakasgn.c, 11
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Description of the block of memory that is about to be lost, including its size and the line number at which it was allocated.
4. Stack trace showing the function call sequence leading to the error.

## Repair

In many cases, this problem is caused by simply forgetting to free a previously allocated block of memory when a pointer is reassigned. For example, the leak in the example code can be corrected as follows:

```
10:      b = (char *)malloc(10);
11:      free(b);
12:      b = a;
```

Some applications may be unable to free memory blocks and may not need to worry about their permanent loss. To suppress these error messages, suppress

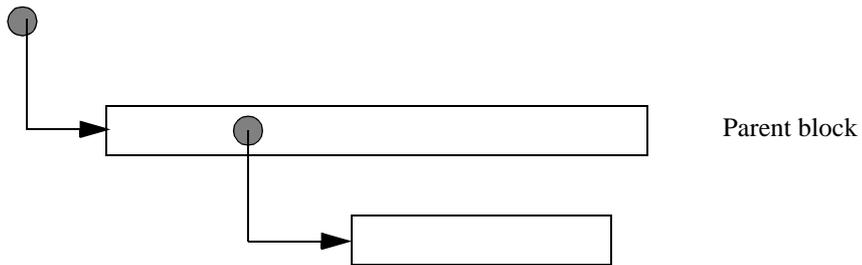
```
LEAK_ASSIGN
```

in the Suppressions Control Panel.

# LEAK\_FREE

## Memory leaked freeing block

This problem can occur when a block of memory contains a pointer to another dynamically allocated block, as indicated in the following figure.



If the main memory block is freed its memory becomes invalid, which means that the included pointer can no longer be used to free the second block. This causes a permanent memory leak.



## Problem

This code defines `PB` to be a data structure that contains a pointer to another block of memory.

```

1:      /*
2:      * File: leakfree.c
3:      */
4:      #include <stdlib.h>
5:
6:      typedef struct ptrblock {
7:          char *ptr;
8:      } PB;
9:
10:     main()
11:     {
12:         PB *p;
13:
14:         p = (PB *)malloc(sizeof(*p));
15:         p->ptr = malloc(10);
16:
17:         free(p);
18:         return (0);
19:     }

```

We first create a single `PB` and then allocate a block of memory for it to point to. The call to `free` on the `PB` then causes a permanent memory leak, since it frees the memory containing the only pointer to the second allocated block. This latter block can no longer be freed.

## Diagnosis (at runtime)

```

[leakfree.c:17] **LEAK_FREE**
1 >>         free(p);

2         Memory leaked freeing block: <return>

3         Lost block:           0x00013888 thru 0x00013891 (10 bytes)
                                block allocated at:
                                    malloc() (interface)
                                        main() leakfree.c, 15
                                Stack trace where the error occurred:
4         main() leakfree.c, 17

```

1. Source line at which the problem was detected.
2. Description of the problem and the value that is about to be lost.

3. Description of the block of memory that is about to be lost, including its size and the line number at which it was allocated.
4. Stack trace showing the function call sequence leading to the error.

## Repair

In many cases, this problem is caused by forgetting to free the enclosed blocks when freeing their container. This can be corrected by adding a suitable call to free the memory before freeing the parent block.

Caution must be used when doing this, however, to ensure that the memory blocks are freed in the correct order. Changing the example in the following manner, for example, would still generate the same error:

```
free(p);  
free(p->ptr);
```

because the blocks are freed in the wrong order. The contained blocks must be freed before their parents, because the memory becomes invalid as soon as it is freed. Thus, the second call to `free` in the above code fragment might fail, because the value `p->ptr` is no longer valid. It is quite legal, for example, for the first call to `free` to have set to zero or otherwise destroyed the contents of its memory block. (Many systems allow the out of order behavior, although it is becoming less portable as more and more systems move to dynamically re-allocated (moveable) memory blocks.)

Some applications may be unable to free memory blocks and may not need to worry about their permanent loss. To suppress these error messages in this case suppress

```
LEAK_FREE
```

in the Suppressions Control Panel.

# LEAK\_RETURN

## Memory leaked by ignoring returned value

This error is generated whenever a function returns a pointer to a block of memory which is then ignored by the calling routine. In this case, the allocated memory block is permanently lost and can never be freed.

### Problem

This code calls the function `gimme`, which returns a memory block that is subsequently ignored by the `main` routine.

```

1:      /*
2:      * File: leakret.c
3:      */
4:      #include <stdlib.h>
5:
6:      char *gimme()
7:      {
8:          return malloc(10);
9:      }
10:
11:     main()
12:     {
13:         gimme();
14:         return (0);
15:     }

```

### Diagnosis (at runtime)

```

[leakret.c:8] **LEAK_RETURN**
1 >>          gimme();

2           Memory leaked ignoring return value: <return>

3           Lost block:          0x000173e8 thru 0x000173f1 (10 bytes)
                                   block allocated at:
                                   malloc() (interface)
                                   gimme() leakret.c, 8
                                   main() leakret.c, 13

           Stack trace where the error occurred:

```

```
main() leakret.c, 13
```

1. Source line at which the problem was detected.
2. Description of the problem and the block that is to be lost.
3. Description of the block of memory that is about to be lost, including its size and the line number at which it was allocated.

## Repair

This problem usually results from an oversight on the part of the programmer, or a misunderstanding of the nature of the pointer returned by a routine. In particular, it is sometimes unclear whether the value returned points to a static block of memory, which will not need to be freed, or a dynamically allocated one, which should be.

Some applications may be unable to free memory blocks and may not need to worry about their permanent loss. To suppress these error messages in this case, suppress

```
LEAK_RETURN
```

in the Suppressions Control Panel.

# LEAK\_SCOPE

## Memory leaked leaving scope

This error is generated whenever a function allocates memory for its own use and then returns without freeing it or saving a pointer to the block in an external variable. The allocated block can never be freed.

## Problem

This code calls the function `gimme`, which allocates a memory block that is never freed.

```

1:      /*
2:      * File: leakscop.c
3:      */
4:      #include <stdlib.h>
5:
6:      void gimme()
7:      {
8:          char *p;
9:          p = malloc(10);
10:         return;
11:     }
12:
13:     main()
14:     {
15:         gimme();
16:         return (0);
17:     }

```

## Diagnosis (at runtime)

```

[leakscop.c:10] **LEAK_SCOPE**
1 >>         return;

2           Memory leaked leaving scope: <return>

3           Lost block:           0x0003870 thru 0x00013879 (10 bytes)
                                   block allocated at:
                                   malloc() (interface)
                                   gimme() leakscop.c, 9
                                   main() leakscop.c, 15

```

```
Stack trace where the error occurred:  
4      gimme()    leakscop.c, 10  
      main() leakscop.c, 15
```

1. Source line at which the problem was detected.
2. Description of the problem and the block that is to be lost.
3. Description of the block of memory that is about to be lost, including its size and the line number at which it was allocated.
4. Stack trace showing the function call sequence leading to the error.

## Repair

This problem usually results from an oversight on the part of the programmer and is cured by simply freeing a block before returning from a routine. In the current example, a call to `free(p)` before line 10 would cure the problem.

A particularly easy way to generate this error is to return from the middle of a routine, possibly due to an error condition arising, without freeing previously allocated data. This bug is easy to introduce when modifying existing code.

Some applications may be unable to free memory blocks and may not need to worry about their permanent loss. To suppress these error messages in this case, suppress

```
LEAK_SCOPE
```

in the Suppressions Control Panel.

# PARAM\_BAD\_RANGE

## Array parameter exceeded range

This error is generated whenever a function parameter is declared as an array, but has more elements than the actual argument which was passed.

### Problem

The following code fragment shows an array declared with one size in the main routine and then used with another in a function.

```

1:      /*
2:      * File: parmrange.c
3:      */
4:      int foo(a)
5:          int a[10];
6:      {
7:          return a[5];
8:      }
9:
10:     int b[5];
11:
12:     main()
13:     {
14:         int a;
15:         a = foo(b);
16:         return (0);
17:     }

```

### Diagnosis (at runtime)

```

[parmrange.c:6] **PARAM_BAD_RANGE**
1 >> {

2     Array parameter exceeded range: a

3         bbbbbbb
           | 20 | 20 |
           ppppppppppp

4     Parameter (p)      0xf7fffb04 thru 0xf7fffb2b (40 bytes)

```

```

                    Actual block (b)                0xf7fffb04 thru 0xf7fffb17
                                                    (20 bytes, 5 elements)
5                b, declared at parmrange.c, 10
    
```

```

                    Stack trace where the error occurred:
                    foo() parmrange.c, 6
6                main() parmrange.c, 15
    
```

1. Source line at which the problem was detected.
2. Description of the problem and the name of the parameter that is in error.
3. Schematic showing the relative layout of the memory block which was actually passed as the argument (b) and expected parameter (p). (See “Overflow diagrams” on page 197.)
4. Description of the memory range occupied by the parameter, including its length.
5. Description of the actual block of data corresponding to the argument, including its address range and size. Also includes the name of the real variable which matches the argument and the line number at which it was declared.
6. Stack trace showing the function call sequence leading to the error.

## Repair

This error is normally easy to correct based on the information presented in the diagnostic output.

The simplest solution is to change the definition of the array in the called routine to indicate an array of unknown size, i.e., replace line 5 with

```
parmrange.c, 5                int a[];
```

This declaration will match any array argument and is the recommended approach whenever the called routine will accept arrays of variable size.

An alternative is to change the declaration of the array in the calling routine to match that expected. In this case, line 10 could be changed to

```
parmrange.c, 10                int b[10];
```

which now matches the argument declaration.

# PARAM\_DANGLING

## Array parameter is dangling pointer

This error is generated whenever a parameter declared as an array is actually passed a pointer to a block of memory that has been freed.

### Problem

The following code frees its memory block before passing it to `foo`.

```

1:      /*
2:      * File: parmdngl.c
3:      */
4:      #include <stdlib.h>
5:
6:      char foo(a)
7:          char a[10];
8:      {
9:          return a[0];
10:     }
11:
12:     main()
13:     {
14:         char *a;
15:         a = malloc(10);
16:         free(a);
17:         foo(a);
18:         return (0);
19:     }

```

### Diagnosis (at runtime)

```

[parmdngl.c:8] **PARAM_DANGLING**
1 >> {
2     Array parameter is dangling pointer: a
3     Pointer   : 0x0001adb0
4     In block  : 0x0001adb0 thru 0x0001adb9 (10 bytes)
                block allocated at:
                    malloc() (interface)
                    main() parmdngl.c, 15

```

```

                    stack trace where memory was freed:
5                    main() parmdngl.c, 16

                    Stack trace where the error occurred:
                    foo() parmdngl.c, 8
6                    main() freedngl.c,17

```

1. Source line at which the problem was detected.
2. Description of the problem and the parameter that is in error.
3. Value of the pointer that was passed and has been deallocated.
4. Information about the block of memory addressed by this pointer, including information about where this block was allocated.
5. Indication of the line at which this block was freed.
6. Stack trace showing the function call sequence leading to the error.

## Repair

This error is normally caused by freeing a piece of memory too soon. A good strategy is to examine the line of code indicated by the diagnostic message which shows where the memory block was freed and check that it should indeed have been de-allocated.

A second check is to verify that the correct parameter was passed to the subroutine.

A third strategy which is sometimes useful is to `NULL` pointers that have been freed and then check in the called subroutine for this case. Code similar to the following is often useful

```

#include <stdlib.h>

char foo(a)
    char *a;
{
    if(a) return a[0];
    return '!';
}

main()
{

```

```
char *a;  
a = (char *)malloc(10);  
free(a);  
a = NULL;  
foo(a);  
return (0);  
}
```

The combination of resetting the pointer to `NULL` after freeing it and the check in the called subroutine prevents misuse of dangling pointers.

# PARAM\_NULL

## Array parameter is NULL

This error is generated whenever a parameter declared as an array is actually passed a NULL pointer.

### Problem

The following code fragment shows a function which is declared as having an array parameter, but which is invoked with a NULL pointer. The value of `array` is NULL because it is a global variable, initialized to zero by default.

```

1:      /*
2:      * File: parmnull.c
3:      */
4:      int foo(a)
5:          int a[];
6:      {
7:          return 12;
8:      }
9:
10:     int *array;
11:
12:     main()
13:     {
14:         foo(array);
15:         return (0);
16:     }

```

### Diagnosis (at runtime)

```

[parmnull:6] **PARAM_NULL**
1 >> {

2     Array parameter is null: a
3     Stack trace where the error occurred:
4         foo() parmnull.c, 6
5         main() parmnull.c, 14

```

1. Source line at which the problem was detected.

2. Description of the problem and the name of the parameter that is in error.
3. Stack trace showing the function call sequence leading to the error.

## Repair

A common cause of this error is the one given in this example, a global pointer which is initialized to zero by the compiler and then never reassigned. The correction for this case is to include code to initialize the pointer, possibly by allocating dynamic memory or by assigning it to some other array object.

For example, we could change the `main` routine of the example to

```
main()
{
    int local[10];

    array = local;
    foo(array);
}
```

This problem can also occur when a pointer is set to `NULL` by the code (perhaps to indicate a freed block of memory) and then passed to a routine that expects an array as an argument.

In this case, Insure++ distinguishes between functions whose arguments are declared as arrays

```
int foo(int a[])
{
```

and those with pointer arguments

```
int foo(int *a)
{
```

The latter type will not generate an error if passed a `NULL` argument, while the former will.

A final common problem is caused when one of the dynamic memory allocation routines, `malloc`, `calloc`, or `realloc`, fails and returns a `NULL` pointer. This can happen either because your program passes bad arguments or simply because it asks for too much memory. A simple way of finding this problem with Insure++ is to enable the `RETURN_FAILURE`

## PARM\_NULL

error code (see “RETURN\_FAILURE” on page 324) via your Advanced Options and run the program again. It will then issue diagnostic messages every time a system call fails, including the memory allocation routines.

If your application cannot avoid passing a `NULL` pointer to a routine, you should either change the declaration of its argument to the second style or suppress these error messages by suppressing

`PARM_NULL`

in the Suppressions Control Panel.

# PARAM\_UNINIT\_PTR

## Array parameter is uninitialized pointer

This error is generated whenever an uninitialized pointer is passed as an argument to a function which expects an array parameter.

### Problem

This code passes the uninitialized pointer `a` to routine `foo`.

```

1:      /*
2:      * File: parmuptr.c
3:      */
4:      char foo(a)
5:          char a[10];
6:      {
7:          return a[0];
8:      }
9:
10:     main()
11:     {
12:         char *a;
13:
14:         foo(a);
15:         return (0);
16:     }

```

### Diagnosis (at runtime)

```

[parmuptr.c:6] **PARAM_UNINIT_PTR**
1 >> {

2     Array parameter is uninitialized pointer: a

3     Stack trace where the error occurred:
        foo()    parmuptr.c, 6
        main()  parmuptr.c, 14

```

1. Source line at which the problem was detected.
2. Description of the problem and the argument that is in error.

3. Stack trace showing the function call sequence leading to the error

## Repair

This problem is usually caused by omitting an assignment or allocation statement that would initialize a pointer. The code given, for example, could be corrected by including an assignment as shown below.

```
/*
 * File: parmuptr.c (Modified)
 */
...
main()
{
    char *a, b[10];
    a = b;
    foo(a);
}
```

# PARAM\_WILD

## Array parameter is wild

This error is generated whenever a parameter is declared as an array but the actual value passed when the function is called points to no known memory block.

This can come about in several ways:

- Errors in user code that result in pointers that don't point at any known memory block.
- Compiling only *some* of the files that make up an application. This can result in Insure++ not knowing enough about memory usage to distinguish correct and erroneous behavior.

**Note:** This section focuses on the first type of problem described above. For information about the second type of problem, contact ParaSoft's Quality Consultants.

## Problem #1

The following code attempts to pass the address of a local variable to the routine `foo` but contains an error at line 14 - the address operator (`&`) has been omitted.

```
1:      /*
2:      * File: parmwld1.c
3:      */
4:      void foo(a)
5:          int a[];
6:      {
7:          return;
8:      }
9:
10:     main()
11:     {
12:         int i = 123, *a;
13:
14:         a = i;
15:         foo(a);
16:         return (0);
17:     }
```

## Diagnosis (at runtime)

```
[parmwld1.c:6] **PARM_WILD**
1 >> {
2     Array parameter is wild: a
3     Pointer : 0x0000007b
4
5         Stack trace where the error occurred:
6             foo() parmwld1.c, 6
7             main() parmwld1.c, 15
```

1. Source line at which the problem was detected.
2. Description of the problem and the name of the parameter that is in error.
3. Value of the bad pointer.
4. Stack trace showing the function call sequence leading to the error.

Note that most compilers will generate warning messages for this error since the assignment uses incompatible types.

## Problem #2

A more insidious version of the same problem can occur when using `union` types. The following code first assigns the pointer element of a union but then overwrites it with another element before finally passing it to a function.

```
1:         /*
2:         * File: parmwld2.c
3:         */
4:         union {
5:             int *ptr;
6:             int ival;
7:         } u;
8:
9:         void foo(a)
10:            int a[];
11:         {
```

```

12:             return;
13:         }
14:
15:     main()
16:     {
17:         int i = 123;
18:
19:         u.ptr = (int *)&i;
20:         u.ival = i;
21:         foo(u.ptr);
22:         return (0);
23:     }

```

Note that this code will not generate compile time errors.

## Diagnosis (at runtime)

```

[parmwld2.c:11] **PARM_WILD**
1 >> {
2     Array parameter is wild: a
3     Pointer : 0x0000007b
4
           Stack trace where the error occurred:
           foo() parmwld2.c, 11
           main() parmwld2.c, 21

```

1. Source line at which the problem was detected.
2. Description of the problem and the name of the parameter that is in error.
3. Value of the bad pointer.
4. Stack trace showing the function call sequence leading to the error.

## Repair

This problem is most conveniently tracked in a debugger by stopping the program at the indicated source line. You should then examine the illegal value and attempt to see where it was generated. Alternatively you can stop the program at some point prior to the error and single-step through the code leading up to the problem.

PARM\_WILD

Note that wild pointers can also be generated when Insure++ has only partial information about your program's structure. For more information about this topic, contact ParaSoft's Quality Consultants.

# READ\_BAD\_INDEX

## Reading array out of range

This error is generated whenever an illegal value will be used to index an array. It is a particularly common error that can be very difficult to detect, especially if the out-of-range elements happen to have zero values.

If this error can be detected during compilation, an error will be issued instead of the normal runtime error.

## Problem

This code attempts to access an illegal array element due to an incorrect loop range.

```

1:      /*
2:      * File: readindx.c
3:      */
4:      int a[10];
5:      int junk;
6:      main()
7:      {
8:          int i, tot=0;
9:
10:         for(i=1; i<=10; i++)
11:             tot += a[i];
12:         return (0);
13:     }
```

## Diagnosis (at runtime)

```

[readindx.c:11] **READ_BAD_INDEX**
1 >>         tot += a[i];

2           Reading array out of range: a[i]

3           Index used: 10

4           Valid range: 0 thru 9 (inclusive)

           Stack trace where the error occurred:
5           main() readindx.c, 11
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Illegal index value used.
4. Valid index range for this array.
5. Stack trace showing the function call sequence leading to the error.

## Repair

Typical sources of this error include loops with incorrect initial or terminal conditions, as in this example, for which the corrected code is:

```
main()
{
    int i, tot=0, a[10];

    for(i=0; i<sizeof(a)/sizeof(a[0]); i++)
        tot += a[i];
    return (0);
}
```

# READ\_DANGLING

## Reading from a dangling pointer

This problem occurs when an attempt is made to dereference a pointer that points to a block of memory that has been freed.

### Problem

This code attempts to use a piece of dynamically allocated memory after it has already been freed.

```

1:  /*
2:   * File: readdngl.c
3:   */
4:  #include <stdlib.h>
5:
6:  main()
7:  {
8:      char b;
9:      char *a = (char *)malloc(10);
10:
11:     free(a);
12:     b = *a;
13:     return (0);
14: }
```

### Diagnosis (at runtime)

```

[readdngl.c:12] **READ_DANGLING**
1 >>          b = *a;

2          Reading from a dangling pointer: a

3          Pointer: 0x000173e8
4          In block: 0x000173e8 thru 0x000173f1 (10 bytes)
           block allocated at:
                   malloc() (interface)
                   main() readdngl.c, 9

5          stack trace where memory was freed:
```

```
main() readdngl.c, 11
```

```
Stack trace where the error occurred:
```

```
6 main() readdngl.c, 12
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Value of the dangling pointer variable.
4. Description of the block to which this pointer used to point, including its size, name and the line at which it was allocated.
5. Stack trace showing where this block was freed.
6. Stack trace showing the function call sequence leading to the error.

## Repair

Check that the de-allocation that occurs at the indicated location should, indeed, have taken place. Also check that the pointer you are using should really be pointing to a block allocated at the indicated place.

# READ\_NULL

## Reading NULL pointer

This error is generated whenever an attempt is made to dereference a `NULL` pointer.

### Problem

This code attempts to use a pointer which has not been explicitly initialized. Since the variable `a` is global, it is initialized to zero by default, which results in dereferencing a `NULL` pointer in line 10.

```

1:      /*
2:      * File: readnull.c
3:      */
4:      int *a;
5:
6:      main()
7:      {
8:          int b, c;
9:
10:         b = *a;
11:     }
```

### Diagnosis (at runtime)

```

[readnull.c:10] **READ_NULL**
1 >>          b = *a;

2           Reading null pointer: a
           Stack trace where the error occurred:
3             main() readnull.c, 10

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

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Stack trace showing the function call sequence leading to the error.

4. Informational message indicating that a serious error has occurred which may cause the program to crash.

## Repair

A common cause of this problem is the one shown in the example - use of a pointer that has not been assigned and which is initialized to zero. This is usually due to the omission of an assignment or allocation statement which would give the pointer a reasonable value.

The example code might, for example, be corrected as follows:

```
1:      /*
2:      * File: readnull.c (modified)
3:      */
4:      int *a;
5:
6:      main()
7:      {
8:          int b, c;
9:
10:         a = &c;
11:         b = *a;
12:     }
```

A second common source of this error is code which dynamically allocates memory, but then zeroes pointers as blocks are freed. In this case, the error would indicate reuse of a freed block.

A final common problem is caused when one of the dynamic memory allocation routines, `malloc`, `calloc`, or `realloc`, fails and returns a `NULL` pointer. This can happen either because your program passes bad arguments or simply because it asks for too much memory. A simple way of finding this problem with `Insure++` is to enable the `RETURN_FAILURE` error code (see “`RETURN_FAILURE`” on page 324) through the Suppressions Control Panel and run the program again. It will then issue diagnostic messages every time a system call fails, including the memory allocation routines.



```
memcpy() (interface)
main() readovr1.c, 9
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Schematic showing the relative layout of the actual memory block (b) and region being read (r) (see “Overflow diagrams” on page 197).
4. Range of memory being read and description of the block from which the read is taking place, including its size and the location of its declaration.
5. Stack trace showing the function call sequence leading to the error.

## Problem #2

A second fairly common case arises when strings are not terminated properly. The code shown below copies a string using the `strncpy` routine, which leaves it non-terminated since the buffer is too short. When we attempt to print this message, an error results.

```
1:      /*
2:      * File: readovr2.c
3:      */
4:      main()
5:      {
6:          char junk;
7:          char b[8];
8:          strncpy(b, "This is a test",
9:                sizeof(b));
10:         printf("%s\n", b);
11:         return (0);
12:     }
```

## Diagnosis (at runtime)

```
[readovr2.c:10] **READ_OVERFLOW**
1 >>         printf("%s\n", b);

2         String is not null terminated within range: b
```

```

3      Reading           : 0xf7fffb50
4      From block:      0xf7fffb50 thru 0xf7fffb57 (8 bytes)
                          b, declared at readovr2.c, 7

      Stack trace where the error occurred:
5      main() readovr2.c, 10

```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Pointer being used as a string.
4. Block from which the read is taking place, including its size and the location of its declaration.
5. Stack trace showing the function call sequence leading to the error.

A slight variation on this misuse of strings occurs when the pointer, passed as a string, lies completely outside the range of its buffer. In this case, the diagnostics will appear as above except that the description line will contain the message

```
Alleged string does not begin within legal range
```

### Problem #3

This code attempts to read past the end of the allocated memory block by reading the second element of the union.

```

1:      /*
2:      * File: readovr3.c
3:      */
4:      #include <stdlib.h>
5:
6:      struct small {
7:          int x;
8:      };
9:
10:     struct big {
11:         double y;
12:     };
13:
14:     union two
15:     {

```

```

16:         struct small a;
17:         struct big b;
18:     };
19:
20:     int main()
21:     {
22:         struct small *var1;
23:         union two *ptr;
24:         double d;
25:
26:         var1 = (struct small *)malloc (sizeof(struct small));
27:         ptr = (union two *) var1;
28:         d = ptr->b.y;
29:         return (0);
30:     }

```

## Diagnosis (at runtime)

```

[readovr3.c:28] **READ_OVERFLOW**
1 >>         d = ptr->b.y;

2         Structure reference out of range: ptr

3         bbbbb
         | 4 | 4 |
         rrrrrrrrr

4         Reading (r):         0x0001fce0 thru 0x0001fce7 (8 bytes)
         From block(b):       0x0001fce0 thru 0x0001fce3 (4 bytes)
         block allocated at:
                                 malloc() (interface)
                                 main() readovr3.c, 26

5         Stack trace where the error occurred:
                                 main() readovr3.c, 28

```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Schematic showing the relative layout of the actual memory block (b) and region being read (r). (See “Overflow diagrams” on page 197.)

4. Range of memory being read and description of the block from which the read is taking place, including its size and the location of its declaration.
5. Stack trace showing the function call sequence leading to the error.

## Problem #4

This code shows a C++ problem that can occur when using inheritance and casting pointers incorrectly.

```

1:      /*
2:      * File: readover.cpp
3:      */
4:      #include <stdlib.h>
5:
6:      class small
7:      {
8:      public:
9:          int x;
10:     };
11:
12:     class big : public small
13:     {
14:     public:
15:         double y;
16:     };
17:
18:     int main()
19:     {
20:         small *var1;
21:         big *var2;
22:         double d;
23:
24:         var1 = new small;
25:         var2 = (big *) var1;
26:         d = var2->y;
27:         return (0);
28:     }

```

## Diagnosis (at runtime)

```

[readover.cpp:26] **READ_OVERFLOW**
1 >>          d = var2->y;

```

```

2      Structure reference out of range: var2

          bbbbb
3      | 4 | 4 | 8 |
          rrrrrrr

      Reading (r):      0x0001fce0 thru 0x0001fce7 (8 bytes)
4      From block(b):  0x0001fce0 thru 0x0001fce3 (4 bytes)
                          var1, allocated at:
                          operator new()
                          main() readover.cpp, 24
      Stack trace where the error occurred:
5      main() readover.cpp, 26

```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Schematic showing the relative layout of the actual memory block (b) and region being read (r). (See “Overflow diagrams” on page 197.)
4. Range of memory being read and description of the block from which the read is taking place, including its size and the location of its declaration.
5. Stack trace showing the function call sequence leading to the error.

## Repair

These errors often occur when reading past the end of a string or using the `sizeof` operator incorrectly. In most cases, the indicated source line contains a simple error.

The code for problem #1 could, for example, be corrected by changing line 9 to

```
memcpy(b, a, strlen(a)+1);
```

# READ\_UNINIT\_MEM

## Reading uninitialized memory

The use of uninitialized memory is a difficult problem to isolate, since the effects of the problem may not show up until much later. This problem is complicated by the fact that quite a lot of references to uninitialized memory are harmless.

To deal with these issues, Insure++ distinguishes two sub-categories of the `READ_UNINIT_MEM` error class

- `copy` - This error code is generated whenever an application assigns a variable using an uninitialized value. In itself, this may not be a problem, since the value may be reassigned to a valid value before use or may never be used. This error category is suppressed by default.
- `read` - This code is generated whenever an uninitialized value is used in an expression or some other context where it must be incorrect. This error category is enabled by default, but is detected only if the `checking_uninit` option is `on`. (see “Options used by Insure++” on page 179)

The difference between these two categories is illustrated in the following examples.

**Note:** Full checking may be disabled by setting the option `checking_uninit off` (see “Options used by Insure++” on page 179) in your Advanced Options. If full uninitialized memory checking is disabled, uninitialized pointers will still be detected, but will be reported in the `READ_UNINIT_PTR` category (see “`READ_UNINIT_PTR`” on page 317).

## Problem #1

This code attempts to use a structure element which has never been initialized.

```
1:      /*
2:      * File: readunil.c
3:      */
4:      #include <stdio.h>
```

```

5:
6:     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:    }

```

## Diagnosis (at runtime)

```

[readunil.c:17] **READ_UNINIT_MEM(read)**
1 >>          area = box.width * box.height;

2           Reading uninitialized memory: box.height
           Stack trace where the error occurred:
3           main() readunil.c, 17

```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Stack trace showing the function call sequence leading to the error.

## Problem #2

This code assigns the value `b` using memory returned by the `malloc` system call, which is uninitialized.

```

1:         /*
2:         * File: readuni2.c
3:         */
4:         #include <stdlib.h>
5:
6:         main()
7:         {
8:             int *a = (int *)malloc(5);

```

```

9:             int b;
10:
11:             b = *a;
12:             return (0);
13:         }

```

The code in line 11 of this example falls into the `copy` error sub-category, since the uninitialized value is merely used to assign another variable. If `b` were later used in an expression, it would then generate a `READ_UNINIT_MEM(read)` error.

**Note:** If the `ints` in lines 8 and 9 of the above example were replaced by `chars`, the error would not be detected by default. To see the error in the new example, you would need to set the Advanced Option `checking_uninit_min_size 1`. For more information about this option, see “Options used by Insure++” on page 179.

## Diagnosis (at runtime)

```

[readuni2.c:11] **READ_UNINIT_MEM(copy)**
1 >>             b = *a;

Reading uninitialized memory: *a

In block: 0x00062058 thru 0x0006205c (5 bytes)
block allocated at:
                malloc() (interface)
                main() readuni2.c, 8

Stack trace where the error occurred:
3             main() readuni2.c, 11

```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Stack trace showing the function call sequence leading to the error.

## Repair

As mentioned earlier, the `READ_UNINIT_MEM(copy)` error category is suppressed by default, so you will normally only see errors in the `read` category. In many cases, these will be errors that can be simply corrected by initializing the appropriate variables. In other cases, these values will

READ\_UNINIT\_MEM

have been assigned from other uninitialized variables, which can be detected by unsuppressing the `copy` sub-category and running again.

# READ\_UNINIT\_PTR

## Reading from uninitialized pointer

This error is generated whenever an uninitialized pointer is dereferenced.

**Note:** This error category will be disabled if full uninitialized memory checking is in effect (the default). In this case, errors are detected in the `READ_UNINIT_MEM` category instead. (see “`READ_UNINIT_MEM`” on page 313)

## Problem

This code attempts to use the value of the pointer `a`, even though it has never been initialized.

```

1:      /*
2:      * File: readuptr.c
3:      */
4:      main()
5:      {
6:          int b, *a;
7:
8:          b = *a;
9:          return (0);
10:     }
```

## Diagnosis (at runtime)

```

[readuptr.c:8] **READ_UNINIT_PTR**
1 >>          b = *a;

2          Reading from uninitialized pointer: a

          Stack trace where the error occurred:
3          main() readuptr.c, 8
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Stack trace showing the function call sequence leading to the error.

## Repair

This problem is usually caused by omitting an assignment or allocation statement that would initialize a pointer. The code given can be corrected by including an assignment as shown below.

```
/*
 * File: readuptr.c (Modified)
 */
main()
{
    int b, *a, c;

    a = &c;
    b = *a;
    return (0);
}
```

# READ\_WILD

## Reading wild pointer

This problem occurs when an attempt is made to dereference a pointer whose value is invalid or which Insure++ did not see allocated.

This can come about in several ways:

- Errors in user code that result in pointers that don't point at any known memory block.
- Compiling only *some* of the files that make up an application. This can result in Insure++ not knowing enough about memory usage to distinguish correct and erroneous behavior.

**Note:** This section focuses on the first type of problem described here. For information on the second type of problem, contact ParaSoft's Quality Consultants.

## Problem #1

The following code attempts to use the address of a variable but contains an error at line 8 - the address operator (&) has been omitted.

```

1:      /*
2:      * File: readwld1.c
3:      */
4:      main()
5:      {
6:          int *a, i = 123, b;
7:
8:          a = i;
9:          b = *a;
10:         return (0);
11:     }
```

## Diagnosis (at runtime)

```

[readwld1.c:9] **READ_WILD**
1>>          b = *a;

2          Reading wild pointer: a
```

```

3      Pointer : 0x0000007b

      Stack trace where the error occurred:
4      main() readwld1.c, 9

```

1. Source line at which the problem was detected.
2. Description of the problem and the name of the parameter that is in error.
3. Value of the bad pointer.
4. Stack trace showing the function call sequence leading to the error.

Note that most compilers will generate warning messages for this error since the assignment uses incompatible types.

## Problem #2

A more insidious version of the same problem can occur when using union types. The following code first assigns the pointer element of a union but then overwrites it with another element before using it.

```

1:      /*
2:      * File: readwld2.c
3:      */
4:      union {
5:          int *ptr;
6:          int ival;
7:      } u;
8:
9:      main()
10:     {
11:         int b, i = 123;
12:
13:         u.ptr = &i;
14:         u.ival = i;
15:         b = *u.ptr;
16:         return (0);
17:     }

```

Note that this code will not generate compile time errors.

## Diagnosis (at runtime)

```
[readwld2.c:15] **READ_WILD**  
1 >>          b = *u.ptr;  
  
2          Reading wild pointer: u.ptr  
  
3          Pointer : 0x0000007b  
  
          Stack trace where error occurred:  
4          main() readwld2.c, 15
```

1. Source line at which the problem was detected.
2. Description of the problem and the name of the parameter that is in error.
3. Value of the bad pointer.
4. Stack trace showing the function call sequence leading to the error.

## Repair

The simpler types of problem are most conveniently tracked in a debugger by stopping the program at the indicated source line. You should then examine the illegal value and attempt to see where it was generated. Alternatively you can stop the program at some point shortly before the error and single-step through the code leading up to the problem.

Note that wild pointers can also be generated when Insure++ has only partial information about your program's structure. For more information on this topic, contact ParaSoft's Quality Consultants.

# RETURN\_DANGLING

## Returning pointer to local variable

This error is generated whenever a function returns a pointer to a (non-static) local variable. Since the stack frame of this routine will disappear when the function returns, this pointer is never valid.

## Problem

The following code shows the routine `foo` returning a pointer to a local variable.

```
1:      /*
2:      * File: retdngl.c
3:      */
4:      char *foo()
5:      {
6:          char b[10];
7:          return b;
8:      }
9:
10:     main()
11:     {
12:         char *a = foo();
13:         return 0;
14:     }
```

## Diagnosis (during compilation)

```
1 [retdngl.c:7] **RETURN_DANGLING**
2     Returning pointer to local variable: b.
>>     return b;
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.

## Repair

The pointer returned in this manner can be made legal in one of several ways.

- Passing the required buffer from the calling function, and if required also passing the size of the buffer as yet another parameter
- Declaring the memory block `static` in the called routine, i.e., line 6 would become

```
static char b[10];
```

- Allocating a block dynamically instead of on the stack and returning a pointer to it, e.g.,

```
char *foo()  
{  
    return malloc(10);  
}
```

- Making the memory block into a global variable rather than a local one.

Occasionally, the value returned from the function is never used in which case it is safest to change the declaration of the routine to indicate that no value is returned.

# RETURN\_FAILURE

## Function call returned an error

A particularly difficult problem to track with conventional methods is that of incorrect return code from system functions. Insure++ is equipped with interface definitions for system libraries that enable it to check for errors when functions are called. Normally, these messages are suppressed, since applications often include their own handling for system calls that return errors. In some cases, however, it may be useful to enable these messages to track down totally unexpected behavior.

## Problem

A particularly common problem occurs when applications run out of memory as in the following code.

```

1:      /*
2:      * File: retfail.c
3:      */
4:      #include <stdlib.h>
5:
6:      main()
7:      {
8:          char *p;
9:
10:         p = malloc(1024*1024*1024);
11:         return (0);
12:     }
```

## Diagnosis

Normally, this code will run without displaying any messages. If RETURN\_FAILURE messages are enabled, however, the following display will result.

```

[retfail.c:10] **RETURN_FAILURE**
1 >>         p = malloc(1024*1024*1024);

           Function returned an error:
2         malloc(1073741824) failed: no more memory
```

```
3      Stack trace where the error occurred:  
      malloc() (interface)  
      main() retfail.c, 10
```

1. Source line at which the problem was detected.
2. Description of the error and the parameters used.
3. Stack trace showing the function call sequence leading to the error.

## Repair

These messages are normally suppressed, but can be enabled by unsuppressing

```
RETURN_FAILURE
```

in the Suppressions Control Panel.

# RETURN\_INCONSISTENT

## Function has inconsistent return type

Insure++ checks that each function returns a result consistent with its declared data type, and that a function with a declared return type actually returns an appropriate value.

Because there are several different ways in which functions and return values can be declared, Insure++ divides up this error category into four levels or subcategories as follows:

- Level 1 - Function has no explicitly declared return type (and so defaults to `int`) and returns no value. (This error level is normally suppressed.)
- Level 2 - Function is explicitly declared to return type `int` but returns nothing.
- Level 3 - Function explicitly declared to return a data type other than `int` but returns no value.
- Level 4 - The function returns the value for one data type at one statement and another data type at another statement.

In many applications, errors at levels 1 and 2 need to be suppressed, since older codes often include these constructs.

## Problem

The following code demonstrates the four different error levels.

```
1:      /*
2:      * File: retinc.c
3:      */
4:      func1() {
5:          return;
6:      }
7:
8:      int func2() {
9:          return;
10:     }
11:
12:     double func3() {
```

```

13:             return;
14:         }
15:
16:     int func4(a)
17:     {
18:         int a;
19:         if (a < 3) return a;
20:         return;
21:     }

```

## Diagnosis (During compilation)

```

1 [retinc.c:4] **RETURN_INCONSISTENT(1)**
2     Function func1 has an inconsistent return type.
   Declared return type implicitly "int",
   but returns no value.
>> func1() {
[retinc.c:8] **RETURN_INCONSISTENT(2)**
   Function func2 has an inconsistent return type.
   Declared return type "int", but returns no value.
>> int func2() {
[retinc.c:12] **RETURN_INCONSISTENT(3)**
   Function func2 has an inconsistent return type.
   Declared return type "double", but returns no value.
>> double func3() {
[retinc.c:20] **RETURN_INCONSISTENT(4)**
   Function func4 has an inconsistent return type.
   Returns value in one location, and not in another.
>> return;

```

1. Source line at which the problem was detected.
2. Description of the error and the parameters used.

## Repair

As already suggested, older codes often generate errors at levels 1 and 2 which are not particularly serious. You can either correct these problems by adding suitable declarations or suppress them by suppressing

```
RETURN_INCONSISTENT(1, 2)
```

in the Suppressions Control Panel.

Errors at levels 3 and 4 should probably be investigated and corrected.

# UNUSED\_VAR

## Unused variables

Insure++ has the ability to detect unused variables in your code. Since these are not normally errors, but informative messages, this category is disabled by default.

Two different sub-categories are distinguished.

- `assigned` - The variable is assigned a value but never used.
- `unused` - The variable is never used.

## Problem #1

The following code assigns a value to the variable `max` but never uses it.

```

1:      /*
2:      * File: unuassign.c
3:      */
4:      main()
5:      {
6:          int i, a[10];
7:          int max;
8:
9:          a[0] = 1;
10:         a[1] = 1;
11:         for(i=2; i<10; i++)
12:             a[i] = a[i-1]+a[i-2];
13:         max = a[9];
14:     }
```

## Diagnosis (during compilation)

Normally this code will run without displaying any messages. If `UNUSED_VAR` messages are enabled, however, the following display will result.

```

1 [unuassign.c:7] **UNUSED_VAR(assigned)**
2     Variable assigned but never used: max
>>     int max;
```

1. Source line at which the problem was detected.
2. Description of the error and the parameters used.

## Problem #2

The following code never uses the variable `max`.

```

1:      /*
2:      * File: unuvar.c
3:      */
4:      main()
5:      {
6:          int i, a[10];
7:          int max;
8:
9:          a[0] = 1;
10:         a[1] = 1;
11:         for(i=2; i<10; i++)
12:             a[i] = a[i-1]+a[i-2];
13:     }
```

## Diagnosis (during compilation)

If `UNUSED_VAR` messages are enabled, however, the following display will result.

```

1 [unuvar.c:7] **UNUSED_VAR(used)**
2     Variable declared but never used: max
>>     int max;
```

1. Source line at which the problem was detected.
2. Description of the error and the parameters used.

## Repair

These messages are normally suppressed but can be enabled by unsuppressing

```
UNUSED_VAR
```

in the Suppressions Control Panel.

You can also enable each sub-category independently by unsuppressing

## UNUSED\_VAR

`UNUSED_VAR(assigned)`

In most cases, the corrective action to be taken is to remove the offending statement, since it is not affecting the behavior of the application. In certain circumstances, these errors might denote logical program errors in which a variable should have been used but wasn't.

# USER\_ERROR

## User generated error message

This error is generated when a program violates a rule specified in an interface module. These normally check that parameters passed to system level or user functions fall within legal ranges or are otherwise valid. This behavior is different from the `RETURN_FAILURE` error code, which normally indicates that the call to the function was made with valid data, but that it still returned an error for some, possibly anticipated, reason.

## Problem

These problems fall into many different categories. A particularly simple example is shown in the following code, which calls the `sqrt` function and passes it a negative argument.

```
1:      /*
2:      * File: usererr.c
3:      */
4:      #include <math.h>
5:
6:      main()
7:      {
8:          double q;
9:
10:         q = sqrt(-2.0);
11:         return (0);
12:     }
```

## Diagnosis (at runtime)

```
[usererr.c:10] **USER_ERROR**
1 >>         q = sqrt(-2.0);

2           Negative number -2.000000 passed to sqrt:

           Stack trace where the error occurred:
3           main() usererr.c, 10
```

## USER\_ERROR

1. Source line at which the problem was detected.
2. Description of the error and the parameters used.
3. Stack trace showing the function call sequence leading to the error.

### Repair

Each message in this category is caused by a different problem, which should be evident from the printed diagnostic. Usually, these checks revolve around the legality of various arguments to functions.

These messages can be suppressed by suppressing

`USER_ERROR`

in the Suppressions Control Panel

# VIRTUAL\_BAD

## Error in runtime initialization of virtual functions

This error is caused when a virtual function has not been initialized prior to being used by another function.

### Problem

The following pieces of code illustrate this error. The virtual function `func` is declared in `virtbad1.cpp` in the `goo` class. A static variable of this class, `barney`, is also declared in that file. The function `crash` calls `func` through `barney` in line 23. In file `virtbad2.cpp`, a static variable of class `foo`, `fred`, is declared. Class `foo` calls `crash`, which then in turn ends up calling the virtual function `func`. A virtual function's address is not established until the program is initialized at runtime, and static functions are also initialized at runtime. This means that depending on the order of initialization, `fred` could be trying to find `func`, which does not yet have an address. The `VIRTUAL_BAD` error message is generated when this code is compiled with Insure++.

**Note:** Due to differences in the object layout of different compilers, this error might not be detected with certain compilers.

```

1:      /*
2:      * File: virtbad1.cpp
3:      */
4:      #include <iostream>
5:
6:      class goo {
7:      public:
8:          int i;
9:          goo::goo() {
10:             cerr << "goo is initialized."
11:                 << endl; }
12:
13:          virtual int func();
14:          virtual int func2();
15:      };
16:      static goo barney;
17:      int crash() {
18:          int ret;
19:          cerr << "Sizeof(goo) = " <<

```

```

18:                                     sizeof(goo) << endl;
19:         cerr << "Sizeof(i) = " <<
20:                                     sizeof(int) << endl;
21:         char *cptr = (char *) &barney;
22:         cptr += 4;
23:         long *lptr = (long *) cptr;
24:         cerr << "vp = " << *lptr << endl;
25:         ret = barney.func();
26:         cerr << "crash" << endl;
27:         return ret;
28:     }
29:     int goo::func() {
30:         cerr << "goo.func" << endl;
31:         func2();
32:         return i;
33:     }
34:     int goo::func2() {
35:         cerr << "goo.func2" << endl;
36:         return 2;
37:     }

```

**Figure 1.** virtbad1.C

```

1:     /*
2:     * File: virtbad2.cpp
3:     */
4:     #include <iostream>
5:
6:     extern int crash();
7:
8:     class foo {
9:     public:
10:        foo::foo() {
11:            cerr << "foo" << endl;
12:            cerr << "Got " <<
13:                crash() << endl;
14:        }
15:    };
16:    static foo fred;

```

**Figure 2.** virtbad2.C

```

1:     /*
2:     * File: virtbad3.cpp

```

```

3:     */
4:     #include <iostream>
5:
6:     int main() {
7:         cerr << "main" << endl;
8:         return 0;
9:     }

```

**Figure 3.** virtbad3.C

## Diagnosis (at runtime)

```

[virtbad1.cpp:29] **VIRTUAL_BAD**
1 >>         func2();

2         Virtual function table is invalid: func2()

3         Stack trace where the error occurred:
           goo::func()virtbad1.cpp, 29
           crash()   virtbad1.cpp, 23
           foo::foo()virtbad2.cpp, 12
           __mod_I__fred0virtbad21001_cc_000()
           _main()
           main()   virtbad3.cpp, 6

**Memory corrupted. Program may crash!**
4 Abort (core dumped)

```

1. Source line at which the problem was detected.
2. Description of the problem and which virtual function caused the error.
3. Stack trace showing the function call sequence leading to the error.
4. Core dumps typically follow these messages, as any usage of the dynamic memory functions will be unable to cope.

## Repair

The error in the sample code could be eliminated by not making `fred` static. In that case, the address for `func` would be generated during the initialization before any requests for it existed, and then no problems would occur.

# WRITE\_BAD\_INDEX

## Writing array out of range

This error is generated whenever an illegal value will be used to index an array which is being written.

If this error can be detected during compilation, a compilation error will be issued instead of the normal runtime error.

## Problem

This code attempts to access an illegal array element due to an incorrect loop range.

```
1:      /*
2:      * File: writindx.c
3:      */
4:      main()
5:      {
6:          int i, a[10];
7:
8:          for(i=1; i<=10; i++)
9:              a[i] = 0;
10:         return (0);
11:     }
```

## Diagnosis (at runtime)

```
[writindx.c:9] **WRITE_BAD_INDEX**
1 >>          a[i] = 0;

2          Writing array out of range: a[i]

3          Index used: 10

4          Valid range: 0 thru 9 (inclusive)
          Stack trace where the error occurred:
5              main() writindx.c, 9

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

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Illegal index value used.
4. Valid index range for this array.
5. Stack trace showing the function call sequence leading to the error.
6. Informational message indicating that a serious error has occurred which may cause the program to crash.

## Repair

This is normally a fatal error and is often introduced algorithmically.

Other typical sources include loops with incorrect initial or terminal conditions, as in this example, for which the corrected code is:

```
main()
{
    int i, a[10];

    for(i=; i<sizeof(a)/sizeof(a[0]); i++)
        a[i] = 0;
    return (0);
}
```

# WRITE\_DANGLING

## Writing to a dangling pointer

This problem occurs when an attempt is made to dereference a pointer that points to a block of memory that has been freed.

### Problem

This code attempts to use a piece of dynamically allocated memory after it has already been freed.

```

1:      /*
2:      * File: writdngl.c
3:      */
4:      #include <stdlib.h>
5:
6:      main()
7:      {
8:          char *a = (char *)malloc(10);
9:
10:         free(a);
11:         *a = 'x';
12:         return (0);
13:     }

```

### Diagnosis (at runtime)

```

[writdngl.c:11] **WRITE_DANGLING**
1 >>         *a = 'x';

2           Writing to a dangling pointer: a

3           Pointer: 0x000173e8
4           In block: 0x000173e8 thru 0x000173f1 (10 bytes)
           block allocated at:
                   malloc() (interface)
                   main() writdngl.c, 8
5           stack trace where memory was freed:
                   main() writdngl.c, 10
6           Stack trace where the error occurred:
                   main() writdngl.c, 11

```

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

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Value of the dangling pointer variable
4. Description of the block to which this pointer used to point, including its size, name, and the line at which it was allocated.
5. Indication of the line at which this block was freed.
6. Stack trace showing the function call sequence leading to the error.

## Repair

Check that the de-allocation that occurs at the indicated location should indeed have taken place. Also check that the pointer you are using should really be pointing to a block allocated at the indicated place.

# WRITE\_NULL

## Writing to a NULL pointer

This error is generated whenever an attempt is made to dereference a `NULL` pointer.

### Problem

This code attempts to use a pointer which has not been explicitly assigned. Since the variable `a` is global, it is initialized to zero by default, which results in dereferencing a `NULL` pointer in line 8.

```

1:      /*
2:      * File: writnull.c
3:      */
4:      int *a;
5:
6:      main()
7:      {
8:          *a = 123;
9:          return (0);
10:     }

```

### Diagnosis (at runtime)

```

[writnull.c:8] **WRITE_NULL**
1 >>          *a = 123;

2           Writing to a null pointer: a
           Stack trace where the error occurred:
3           main() writnull.c, 8

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

```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Stack trace showing the function call sequence leading to the error.

4. Informational message indicating that a serious error has occurred which may cause the program to crash.

## Repair

A common cause of this problem is the one shown in the example-- use of a pointer that has not been explicitly assigned and which is initialized to zero. This is usually due to the omission of an assignment or allocation statement which would give the pointer a reasonable value.

The example code might, for example, be corrected as follows

```
1:      /*
2:      * File: writnull.c (Modified)
3:      */
4:      int *a;
5:
6:      main()
7:      {
8:          int b;
9:
10:         a = &b;
11:         *a = 123;
12:         return (0);
13:     }
```

A second common source of this error is code which dynamically allocates memory but then zeroes pointers as blocks are freed. In this case, the error would indicate reuse of a freed block.

A final common problem is caused when one of the dynamic memory allocation routines, `malloc`, `calloc`, or `realloc`, fails and returns a `NULL` pointer. This can happen either because your program passes bad arguments or simply because it asks for too much memory. A simple way of finding this problem with `Insure++` is to enable the `RETURN_FAILURE` error code (see “`RETURN_FAILURE`” on page 324) via your Suppressions Control Panel and run the program again. It will then issue diagnostic messages every time a system call fails, including the memory allocation routines.

# WRITE\_OVERFLOW

## Writing overflows memory

This error is generated whenever a block of memory indicated by a pointer will be written outside its valid range.

## Problem

This code attempts to copy a string into the array `a`, which is not large enough.

```

1:      /*
2:      * File: writover.c
3:      */
4:      main()
5:      {
6:          int junk;
7:          char a[10];
8:
9:          strcpy(a, "A simple test");
10:         return (0);
11:     }

```

## Diagnosis (at runtime)

```

[writover.c:9] **WRITE_OVERFLOW**
1 >>          strcpy(a, "A simple test");

2          Writing overflows memory: a

          bbbbbbbbbb
          |  10  |  4  |
          wwwwwwwwwwwwwww

4          Writing (w): 0xf7ffaf0 thru 0xf7ffb09 (14 bytes)
          To block (b):          0xf7ffaf0 thru 0xf7ffb05 (10 bytes)

          a, declared at writover.c, 7

5          Stack trace where the error occurred:
          strcpy () (interface)
          main() writover.c, 9

```

1. Source line at which the problem was detected.
2. Description of the problem and the incorrect expression.
3. Schematic showing the relative layout of the actual memory block (b) and region being written (w). (See “Overflow diagrams” on page 197.)
4. Range of memory being written and description of the block to which the write is taking place, including its size and the location of its declaration.
5. Stack trace showing the call sequence leading to the error.

## Repair

This error often occurs when working with strings. In most cases, a simple fix is to increase the size of the destination object.

# WRITE\_UNINIT\_PTR

## Writing to an uninitialized pointer

This error is generated whenever an uninitialized pointer is dereferenced.

### Problem

This code attempts to use the value of the pointer `a`, even though it has not been initialized.

```

1:      /*
2:      * File: writuptr.c
3:      */
4:      main()
5:      {
6:          int *a;
7:
8:          *a = 123;
9:          return (0);
10:     }

```

### Diagnosis (at runtime)

```

[writuptr.c:8] **WRITE_UNINIT_PTR**
1 >>          *a = 123;

2          Writing to an uninitialized pointer: a

3          Stack trace where the error occurred:
           main() writuptr.c, 8

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

```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Stack trace showing the function call sequence leading to the error.
4. Informational message indicating that a serious error has occurred which may cause the program to crash.

## Repair

This problem is usually caused by omitting an assignment or allocation statement that would initialize a pointer. The code given, for example, could be corrected by including an assignment as shown below.

```
/*
 * File: writuptr.c (Modified)
 */
main()
{
    int *a, b;

    a = &b;
    *a = 123;
}
```

# WRITE\_WILD

## Writing to a wild pointer

This problem occurs when an attempt is made to dereference a pointer whose value is invalid or which Insure++ did not see allocated.

This can come about in several ways:

- Errors in user code that result in pointers that don't point at any known memory block.
- Compiling only *some* of the files that make up an application. This can result in Insure++ not knowing enough about memory usage to distinguish correct and erroneous behavior.

**Note:** This section focuses on the first type of problem described here. For information about the second type of problem, contact ParaSoft's Quality Consultants.

## Problem #1

The following code attempts to use the address of a variable but contains an error at line 8 - the address operator (&) has been omitted.

```

1:      /*
2:      * File: writwld1.c
3:      */
4:      main()
5:      {
6:          int i = 123, *a;
7:
8:          a = i;
9:          *a = 99;
10:         return (0);
11:     }
```

## Diagnosis (at runtime)

```

[writwld1.c:9] **WRITE_WILD**
1 >>          *a = 99;

2          Writing to a wild pointer: a
```

```

3         Pointer : 0x0000007b
4         Stack trace where the error occurred:
           main() writwld1.c, 9

```

1. Source line at which the problem was detected.
2. Description of the problem and the name of the parameter that is in error.
3. Value of the bad pointer.
4. Stack trace showing the function call sequence leading to the error.

Note that most compilers will generate warning messages for this error since the assignment in line 8 uses incompatible types.

## Problem #2

A more insidious version of the same problem can occur when using union types. The following code first assigns the pointer element of a union but then overwrites it with another element before using it.

```

1:         /*
2:         * File: writwld2.c
3:         */
4:         union {
5:             int *ptr;
6:             int ival;
7:         } u;
8:
9:         main()
10:        {
11:            int i = 123;
12:
13:            u.ptr = &i;
14:            u.ival = i;
15:            *u.ptr = 99;
16:            return (0);
17:        }

```

Note that this code will not generate compile time errors.

## Diagnosis (at runtime)

```
[writwld2.c:15] **WRITE_WILD**  
1>>          *u.ptr = 99;  
  
2          Writing to a wild pointer: u.ptr  
  
3          Pointer : 0x0000007b  
  
4          Stack trace where the error occurred:  
          main() writwld2.c, 15
```

1. Source line at which the problem was detected.
2. Description of the problem and the name of the parameter that is in error.
3. Value of the bad pointer.
4. Stack trace showing the function call sequence leading to the error.

## Repair

The simpler types of problems are most conveniently tracked in a debugger by stopping the program at the indicated source line. You should then examine the illegal value and attempt to see where it was generated. Alternatively, you can stop the program at some point shortly before the error and single-step through the code leading up to the problem.

Note that wild pointers can also be generated when Insure++ has only partial information about your program's structure. For more information about this topic, contact ParaSoft's Quality Consultants.

# Insure++ API

This section lists the Insure++ functions that can be called from an application program.

If you are inserting these routines into your source code, you might want to use the pre-processor symbol `__INSURE__` so that they will only be called when compiling with the appropriate tools, e.g.,

```

/*
 * Code being checked with Insure++
 */
...
/*
 * Disable runtime memory checking
 */
#ifdef __INSURE__
    _Insure_set_option("runtime", "off");
#endif
...
/*
 * Block of code without runtime checking...
 */
...
/*
 * Re-enable runtime checking
 */
#ifdef __INSURE__
    _Insure_set_option("runtime", "on");
#endif

```

In this way you can use the same source code when compiling with or without Insure++.

You will also need to add prototypes for these functions to your code, particularly if you are calling these C functions from C++ code. Make sure that in this case your prototype is properly marked extern "C".

## Control routine

This routine affects the behavior of Insure++ and is normally called from within your source code.

- `void _Insure_printf(char *fmt, [,arg...]);`  
Causes Insure++ to add the given character string to its output.

## Memory block description routines

- `long _Insure_list_allocated_memory(void);`  
Prints a list of all allocated memory blocks and their sizes.  
Returns the total number of bytes allocated.
- `void _Insure_mem_info(void *ptr);`  
Displays all information known about the memory block whose address is `ptr`. For example, the following code

```
#include <stdlib.h>

main()
{
    char *p, buf[128];

    p = malloc(100);
#ifdef __INSURE__
    _Insure_mem_info(buf);
    _Insure_mem_info(p);
#endif
    ...
}
```

might generate the following output

```
Pointer   : 0xf7fff74c (stack)
Offset    : 0 bytes
In block  : 0xf7fff74c thru 0xf7fff7cb (128 bytes)
           buf, declared at foo.c, 4
Pointer   : 0x00024b98 (heap)
Offset    : 0 bytes
In block  : 0x00024b98 thru 0x00024bfb (100 bytes)
           p, allocated at foo.c, 6
```

- `void _Insure_ptr_info(void **ptr);`  
Displays all information about the pointer whose address is passed. For example, the code

```
#include <stdlib.h>

main()
```

```
{
    char *p, buf[128];

    p = malloc(100);
#ifdef __INSURE__
    __Insure_ptr_info(&p);
#endif
    ...
}
```

might generate the following output

```
Pointer   : 0x00024b98 (heap)
Offset    : 0 bytes
In block  : 0x00024b98 thru 0x00024bfb (100 bytes)
           : p, allocated at foo.c, 6
```



# Interface Functions

This section lists the Insure++ specific functions that can be called from Insure++ interface files. The use of these functions is described in the section “Interfaces” on page 113. This description gives only a brief summary of the purpose and arguments of the various functions. Probably the best way to see their purpose is to look at the source code for the interfaces shipped with Insure++, which can be found in subdirectories of the main Insure++ installation directory.

Note that these functions, despite appearances, are not C functions that you can insert into your C code. They can *only* be used in Insure++ interface files to be compiled with `iic`.

## Memory Block Declaration Routines

These functions are used to indicate the usage of memory blocks. They do not actually allocate or free memory.

- `void iic_alloc(void *ptr, unsigned long size);`  
Declares a block of uninitialized heap memory of the given size. (See “Interfaces” on page 113.)
- `void iic_alloca(void *ptr, unsigned long size);`  
Declares a block of data on the stack.
- `void iic_alloci(void *ptr, unsigned long size);`  
Declares a block of initialized heap memory of the given size. (Without the second argument, declares a block the length of the first argument treated as a character string, including the terminating `NULL`.) (See “Interfaces” on page 113.)
- `void iic_allocs(void *ptr, unsigned long size);`  
Declares a pointer to a block of static memory. (Without the second argument, declares a block the length of the first argument treated as a character string, including the terminating `NULL`.) (See “Interfaces” on page 113.)
- `void iic_realloc(void *old, void *new, unsigned long new_size);`

Indicates that the indicated block has been re-allocated and that the contents of the old block should be copied to the new one.

- `void iic_save(void *ptr);`  
Specifies that the indicated block should never be reported to have “leaked”. Normally, this is used when the system will keep track of a memory block, even after all user pointers have gone.
- `void iic_unalloc(void *ptr);`  
Deallocates a block of memory. (See “Interfaces” on page 113.)
- `void iic_unallocs(void *ptr);`  
Undoes the effect of an `iic_allocs`. No error checking is performed on the pointer - the block is simply forgotten.

## Memory Checking Routines

These functions report the appropriate *Insure++* error message if the check fails.

- `void iic_copy(void *to, void *from, unsigned long size);`  
Checks for write and read access to the given number of bytes and indicates that the `from` block will be copied onto the `to` block. (See “Interfaces” on page 113.)
- `void iic_copyattr(void *to, void *from)`  
Copies the attribute properties (e.g., opaque, uninitialized, etc.) from one pointer to another.
- `void iic_dest(void *ptr, unsigned long size);`  
Checks for write access to the `ptr` for `size` number of bytes. (See “Interfaces” on page 113.)
- `void iic_freeable(void *ptr);`  
Checks that the indicated pointer indicates a dynamically allocated block of memory that could be freed.
- `void iic_justcopy(void *to, void *from, unsigned long size);`  
Indicates that the `from` block will be copied onto the `to` block without performing the other `iic_copy` checks.

- `void iic_pointer(void *ptr);`  
Checks that the indicated pointer is valid, without checking anything about the size of the block it points to.
- `void iic_resize(void *ptr, unsigned long newsize);`  
Indicates that the block of memory has changed size.
- `void iic_source(void *ptr, unsigned long size);`  
Checks for read access to `ptr` for `size` bytes. (See “Interfaces” on page 113.) Does no checks for initialization of the block.
- `void iic_sourcei(void *ptr, unsigned long size);`  
Checks for read access to `ptr` for `size` bytes, and also that the memory is initialized. (See “Interfaces” on page 113.)
- `int iic_string(char *ptr, unsigned long size);`  
Checks that the pointer indicates a `NULL` terminated string. (See “Interfaces” on page 113.) If the optional second argument is supplied, the check terminates after at most that number of characters. In either case, the string length is returned, or -1 if some error prevented the string length from being computed.

## Function Pointer Checks

- `void iic_declfunc(void (*f)());`  
Declares that the indicated pointer is a function regardless of appearance or other information.
- `void iic_func(void (*f)());`  
Checks that the indicated pointer is actually a function.

## Printf/scanf Checking

- `void iic_input_format(char *format_string);`  
Indicates that the indicated string and the arguments following it should be checked as though they were a `scanf` style format string. This function should be called from the interface before activating the function being checked.
- `void iic_output_format(char *format_string);`  
Indicates that the indicated string and the arguments

following it should be checked as though they were a `printf` style format string.

- `void iic_post_input_format(int ntokens);`  
This function can be called after an `iic_input_format` check and a call to an input function to check that the indicated number of tokens did not corrupt memory when read. If the argument is omitted, all the tokens from the `iic_input_format` string are checked.
- `int iic_strlenf(char *format_string, ...);`  
Returns the length of the string after substitution of the subsequent arguments, which are interpreted as a `printf` style format string.
- `int iic_vstrlenf(char *format_string, va_list ap);`  
Returns the length of the string after substitution of the argument, which must be the standard type for a variable argument list. The `format_string` argument is interpreted in the normal `printf` style.

## Utility Functions

- `char *iic_c_string(char *string);`  
Converts a string to a format consistent with the C language conventions. Useful for printing error messages.
- `void iic_error(int code, char *format, ...);`  
Generates a message with the indicated error code (either `USER_ERROR` or `RETURN_FAILURE`). (See “Interfaces” on page 113.)
- `void iic_expand_subtype(<typename>, <typetag>);`  
Indicates that the structure or union named `typename` contains an element name `typetag` whose size varies at runtime. Normally used for “stretchy” arrays. (See “Stretchy arrays” on page 76.) For example, if you have the following code, and `a` is a stretchy array,

```
struct test {
char a[1];
```

```
};
```

then the appropriate function call would be:

```
iic_expand_subtype(struct test, a);
```

- `int iic_numargs(void);`  
Returns the number of arguments actually passed to the function.
- `void iic_warning(char *string);`  
Prints the indicated string at compile-time.

## Callbacks

- `iic_body`  
Keyword used in function declarations to indicate that the function for which the interface is being specified will be used as a callback.
- `void iic_callback(void (*f)(), void (*template)());`  
Specifies that the function `f` will be used as a callback, and that whenever it is called its invocation is to be processed as indicated by the previously declared (static) function `template`.
- `void iic_opaque_callback(void (*f)());`  
Specifies that the function `f` will be used as a callback, and that all of its arguments should be treated as opaque whenever it is invoked.

## Variable Arguments

- `__dots__`  
Placeholder for variable arguments (“...”) in an argument list. (See “Interfaces” on page 113.)

## Initialization

- `void iic_startup(void)`  
Function that can be declared in any interface file and which

contains calls to be made before any function in the interface is executed. (See “Interfaces” on page 113.)

## Termination

- `void iic_exit(void)`  
Indicates that the function specified by the interface is going to exit. This allows Insure++ to close its files and perform any necessary cleanup activity before the program terminates.

# Index

## Symbols

%a, filename macro 177  
 %c, error category macro 183  
 %c, filename macro 177  
 %d, date macro 183  
 %D, filename macro 177  
 %d, filename macro 177  
 %f, filename macro 184  
 %F, full pathname macro 184  
 %h, hostname macro 184  
 %l, line number macro 184  
 %n, filename macro 178  
 %p, filename macro 178  
 %p, process ID macro 184  
 %R, filename macro 177  
 %r, filename macro 177  
 %T, filename macro 177  
 %t, filename macro 177  
 %t, time macro 184  
 %V, filename macro 178  
 %v, filename macro 178  
 .ins\_orig file extension 186  
 .psrc options  
     interface\_library 122  
     inuse 136, 140  
     rename\_files 99  
 .psrc options  
     interface\_library 123  
     suppress 134  
 .tql file extension 123, 124  
 .tqs file extension 122  
 .tqs file extension 111  
 .tqs version (%T), in filenames 177  
 .tqs version (%t), in filenames 177  
 <argument #> 199  
 <return> 199  
 \x escape sequence 182  
 \_\_dots\_\_ 129  
 \_\_INSURE\_\_ 97, 349

\_\_INSURE\_\_ pre-processor macro  
     97, 107  
 \_Insure\_cleanup 107  
 \_Insure\_list\_allocated\_memory 98, 101  
 \_Insure\_mem\_info 98, 101  
 \_Insure\_ptr\_info 98, 101  
 \_Insure\_set\_option 97, 349  
 \_Insure\_trace\_enable 104  
 \_Insure\_trap\_error 100, 107

## Numerics

16-bit machines 47  
 32-bit machines 47  
 64-bit machines 47

## A

%a, filename macro 177  
 adjacent memory blocks 37  
 Advanced Options  
     interface\_library 112  
     auto\_expand 179  
     compile time 179–188  
     coverage\_switches 82, 193  
     error\_format 69, 70  
     InSra 194  
     interface\_library 112  
     leak\_sort 81  
     leak\_trace 81  
     report\_file 68  
     runtime 188–194  
     signal\_catch 108  
     signal\_ignore 108  
     source\_path 72  
     stack\_internal 104, 106  
     summarize 81  
     trace 104, 106  
     trace\_banner 105  
     trace\_file 105  
     tracing 104–106  
     unsuppress 76  
 Advanced options

## Index

- suppress\_output 75
- alloc1.c 206
- alloc2.c 207
- Alpha, DEC 47
- anonymous structures/unions 77
- ANSI compilers 49
- API
  - \_Insight\_mem\_info 101
  - \_Insure\_cleanup 107
  - \_Insure\_list\_allocated\_memory 98, 101
  - \_Insure\_mem\_info 98
  - \_Insure\_ptr\_info 98, 101
  - \_Insure\_trace\_annotate 105
  - \_Insure\_trace\_enable 104
  - \_Insure\_trap\_error 100
- architecture (%a), in filenames 177
- architectures 123, 176
- <argument #> 199
- arguments
  - checking ranges 51
  - type checking 49–51
- arrays
  - expandable 76, 179
- auto\_expand, Advanced Options 179
- Automatically expand stretchy arrays 77

## B

- badcast.c 209
- baddecl1.c 47, 211
- baddecl2.c 47, 211
- badform1.c 214
- badform2.c 48, 215
- badform3.c 216
- badform4.c 218
- badint.c 219
- badparm1.c 222
- badparm2.c 222
- badparm4.c 223
- bag.c 116
- bag.h 115

- bag\_i.c 116
- bagi.c 116
- bbbbbbbbbb 197
- big-endian 47
- bitfields 54
- black, color in Inuse 144
- Block frequency 146
- blue, color in Inuse 144
- bounds overflow 37, 197
- breakpoints 107
- bugsfunc.c 99
- building interfaces 117
- built-in
  - functions 181
  - types 182
  - variables 182
- Bus error 107
- byte swapping 47

## C

- %c, error category macro 183
- %c, filename macro 177
- calloc 44
- calloc 125
- Chaperon
  - about 53
  - bitfields 54
  - examples 57
  - requirements and limitations 54
  - states 56
  - with gdb 64
- chunks, memory 80
- client-server programming 68, 70
- CodeWizard 32
- colors, in Inuse 144
- compatible (error sub-category) 214, 221
- compilation time (%d), in filenames 177
- compile time warnings
  - C++ 50
- compiler 179, 180
  - using multiple 123, 176
- compiler (%c), in filenames 177
- compiler built-in

- functions 181
- types 182
- variables 182
- complex data types 126
- configuration files
  - insure 67
- console 67
- contacting ParaSoft 15
- context based error suppression 73
- contributing interface modules 134
- control-C 107
- copy, READ\_UNINIT\_MEM sub-category 45
- copybad.c 226
- copydang.c 228
- copyunin.c 230
- copywild.c 232
- Coverage analysis 163
- coverage\_switches, Advanced Options 82, 193
- cross compiling 186
- ctime 125
- CTRL-C 107
- customer support 126

## D

- %d, date macro 183
- %D, filename macro 177
- %d, filename macro 177
- dangling pointers 43–44, 242
- data representations 47
- date (%d), in error report banners 183
- date and time, on error reports 70
- dbx 99
- deadcode.C 234
- deadcode.c 235
- debuggers
  - using Insure++ with 98
- DEC Alpha 47
- default report style 67
- delmis1.C 237
- delmis2.C 238
- diagrams, memory overflow 197
- directories

- searching for source code 72
- distributed programs 70
- \_\_dots\_\_ 129
- dynamic memory
  - common bugs 43
  - pointers to blocks 39
  - using Insure++'s library 185

## E

- EINTR 52
- emacs, customizing error reports for 69
- enabling error codes 76
- endian-ness 47
- environment variables
  - in filenames 124, 176, 178
- error category (%c), in error report banners 183
- error codes 199–348
  - disabled 200
  - enabled 200
  - enabling 76
  - first occurrence 70
  - suppressing messages 73
  - suppressing messages by context 73
- error report format
  - date (%d macro) 183
  - error category (%c macro) 183
  - filenames (%F macro) 184
  - hostname (%h macro)f 184
  - line number (%l macro) 184
  - pathname (%P macro)f 184
  - process ID (%p macro) 184
  - time (%t macro) 184
- error summaries 77
- error\_format, Advanced Options 69, 70
- errors
  - in system calls 51
- examples
  - alloc1.c 206
  - alloc2.c 207
  - badcast.c 209

baddecl1.c 47, 211  
 baddecl2.c 47, 211  
 badform1.c 214  
 badform2.c 48, 215  
 badform3.c 216  
 badform4.c 218  
 badint.c 219  
 badparm1.c 222  
 badparm2.c 222  
 badparm4.c 223  
 bag.C 116  
 “bugs” summary 78  
 bugsfunc.c 99  
 chaperon 57  
 copybad.C 226  
 copydang.C 228  
 copyunin.C 230  
 copywild.c 232  
 “coverage” summary 82  
 deadcode.C 234  
 deadcode.c 235  
 delmis1.C 237  
 delmis2.C 238  
 expdangl.c 242  
 expnull.c 244  
 exprange.c 240  
 expucmp.c 248  
 expudiff.c 250  
 expuptr.c 246  
 expwld1.c 252  
 expwld2.c 253  
 freebody.c 255  
 freedngl.c 257  
 freeglob.c 259  
 freelocl.c 261  
 freeuptr.c 263  
 freewild.c 265  
 funcbad.c 267  
 funcnull.c 269  
 funcuptr.c 271  
 funcwild.c 273  
 hello.c 35  
 hello2.c 37  
 hello3.c 39  
 hello4.c 43  
 interfaces  
  
 C 113  
 C++ 115  
 leakasgn.c 278  
 leakfree.c 281  
 leakret.c 283  
 “leaks” summary 80  
 leakscop.c 285  
 mymal.c 115  
 mymal\_i.c 114  
 mymaluse.c 113  
 noleak.c 138  
 parmdngl.c 289  
 parmnull.c 292  
 parmrange.c 287  
 parmuptr.c 295  
 parmwld1.c 297  
 parmwld2.c 298  
 readdngl.c 303  
 readindx.c 301  
 readnull.c 305  
 readover.C 311  
 readovr1.c 307  
 readovr2.c 45, 308  
 readovr3.c 309  
 readuni1.c 45, 313  
 readuni2.c 314  
 readuptr.c 317  
 readwld1.c 319  
 readwld2.c 320  
 retdngl.c 322  
 retfail.c 324  
 retinc.c 326  
 slowleak.c 135, 136, 137  
 stretch1.c 76  
 stretch2.c 77  
 trace.C 106  
 unuasgn.c 328  
 unuvar.c 329  
 usererr.c 331  
 virtbad1.C 333  
 virtbad2.C 334  
 virtbad3.C 334  
 warn.c 276  
 writdngl.c 338  
 writindx.c 336  
 writnull.c 340

- writover.c 342
- writuptr.c 344
- writwld1.c 346
- writwld2.c 347
- exception handlers 107, 130
- executable directory (%v), in filenames 178
- executable name (%v), in filenames 178
- execution time (%D), in filenames 177
- expandable arrays 76, 179
- expdangl.c 242
- expnull.c 244
- EXPR\_NULL 212
- exprange.c 240
- expucmp.c 248
- expudiff.c 250
- expuptr.c 246
- expwld1.c 252
- expwld2.c 253
- extensions, *see* file extensions

## F

- %F, full pathname macro 184
- %f, filename macro 184
- file extensions
  - .ins\_orig 186
  - .tqs 122
  - .tqs 111
- file permissions 52
- filenames
  - .tqs version (%T macro) 177
  - .tqs version (%t macro) 177
  - architecture (%a macro) 177
  - compilation time (%d macro) 177
  - compiled with (%c macro) 177
  - executable directory (%v macro) 178
  - executable name (%v macro) 178
  - execution time (%D macro) 177
  - expanding macros in 124, 176
  - Insure++ version (%R macro) 177
  - Insure++ version (%r macro) 177
  - process ID (%p macro) 178

- reports 68
- unique numeric extension (%n macro) 178
- using environment variables 178
- filenames (%f), in error report banners 184
- files
  - limit on open 52
  - non-existent 52
- first error 70
- flexible arrays 76, 179, 184
- fork 68, 70
- fprintf, *see* printf
- free 44, 102
- free 125
- freebody.c 255
- freedngl.c 257
- freeglob.c 259
- freeing memory 43
- freeing memory twice 43
- freeing static memory 44
- freelocl.c 261
- freeuptr.c 263
- freewild.c 265
- fscanf, *see* scanf
- fseek 51
- funcbad.c 267
- funcnull.c 269
- function
  - prototypes 49
- function prototypes, used as interfaces 119
- functions
  - mismatched arguments 49–51
  - pointers to 37
  - return types, inconsistent 326
- funcuptr.c 271
- funcwild.c 273

## G

- g++ 180
- gcc 179, 180
- gdb 64
- getenv 125

## Index

gets checking 48  
global variables 37  
GNU emacs, customizing error reports  
for 69  
green, color in Inuse 144

## H

%h, hostname macro 184  
handlers, signal 107  
Heap history 142, 145  
Heap layout 142, 147  
hello.c 35  
hello2.c 37  
hello3.c 39  
hello4.c 43  
hostname 70  
hostname (%h), in error report banners  
184

## I

I/O 48, 52, 107  
ignoring return value 41  
iic 122, 353  
iic 111  
iic\_alloc 121  
iic\_alloc 124  
iic\_alloci 125  
iic\_allocs 125  
iic\_copy 121  
iic\_copy 124  
iic\_dest 124  
iic\_error 111, 124  
iic\_input\_format 129  
iic\_output\_format 129  
iic\_source 124  
iic\_sourcei 124  
iic\_startup 129  
iic\_string 125  
iic\_strlenf 129  
iic\_unalloc 125  
iic\_warning 276  
iiinfo 123

iiwhich 119–122, 123  
iiwhich 125  
incompatible (error sub-category)  
214, 221  
incompatible declarations 47  
inconsistent return types 326  
.ins\_orig file extension 186  
insight 136  
Insra 67, 186, 192  
    Advanced Options 194  
    insra 186, 192  
insra 186, 192  
Insra, troubleshooting 95  
\_\_INSURE\_\_ pre-processor macro  
107  
insure  
    runtime functions 98  
Insure++  
    number of error messages 42  
    report file 67  
*Insure++*  
    using interface files 112  
insure++ 136  
Insure++ version (%R), in filenames  
177  
Insure++ version (%r), in filenames  
177  
\_Insure\_list\_allocated\_memor  
y 98, 101  
\_Insure\_mem\_info 98, 101  
\_Insure\_ptr\_info 98, 101  
\_Insure\_trap\_error 100  
int vs. long 183  
interface\_library 112  
interface\_library 122  
    for multiple platforms 176  
    PARASOFT variable 176  
interface\_library 112, 123  
    for multiple platforms 124  
interfaces 113–134  
    contributing 134  
    examples  
        C 113  
        C++ 115  
    getting started with iiproto 119  
    getting started with iiwhich 121

- interface functions 353–358
  - strategy for creating 117
  - writing 121
- intermittent errors 45
- interrupted system calls 52
- interrupts 107
- Inuse 34
  - black color in 144
  - blue color in 144
  - green color in 144
  - GUI 141
  - introduction 135
  - red color in 144
  - running 140–154
  - use of color in 144
  - yellow color in 144
- inuse 136, 140

## K

- keyboard interrupt 107

## L

- %l, line number macro 184
- LEAK\_ASSIGN 41
- leak\_combine, Advanced Options 81
- LEAK\_FREE 41
- LEAK\_RETURN 41
- LEAK\_SCOPE 41
- leak\_sort, Advanced Options 81
- leak\_trace, Advanced Options 81
- leakasn.c 278
- leakfree.c 281
- leakret.c 283
- leaks, memory 39–43
- leakscop.c 285
- libraries
  - checking arguments to 51
- line number (%l), in error report banners 184
- little-endian 47
- Loading a report file 170

- local variables 37
- long vs. int 183
- look and feel 141

## M

- machine name 70
- macros, pre-defined 97
- malloc 37, 39, 44, 52
  - using Insure++'s 185
- malloc 125
- memcpy 307
- memory
  - adjacent blocks 37
  - allocation 43
  - blocks containing pointers 41
  - chunks 80
  - corruption 35, 197
  - dynamically allocated 39
  - leaks 39–43
  - overflow 49, 197
  - running out of 42
  - shared 52
  - usage summary 80
  - using uninitialized 45
- merging report files 68
- mismatched arguments 49–51
- multiple return types 326
- multiprocessing 70
- mymal.c 115
- mymal\_i.c 114
- mymaluse.c 113

## N

- %n, filename macro 178
- noleak.c 138
- non-existent files 52
- number of error messages 42

## O

- open file limit 52

## Index

- Options used by TCA ??–196
- orphaned memory 39–43
- other (error sub-category) 214, 221
- out of memory 42
- outstanding, summarize keyword 81
- overflow
  - bounds of object 37
  - diagrams 197
  - memory 49, 197
- overwriting memory 37

## P

- %p, filename macro 178
- %p, process ID macro 184
- parallel processing 68, 70
- ParaSoft
  - contacting 15
- PARM\_BAD\_RANGE
  - overflow diagrams 198
- parmdngl.c 289
- parmnull.c 292
- parmrnge.c 287
- parmuptr.c 295
- parmwld1.c 297
- parmwld2.c 298
- pathname (%F), in error report banners 184
- PC 47
- permissions, file 52
- personal computers 47
- pointer (error sub-category) 221, 225
- pointer reassignment 39
- pointers 37
  - dangling 43–44, 242
  - function 37
  - not equivalent to integers 47
  - NULL 37
  - reusing free'd blocks 43
  - uninitialized 37
  - unrelated 37
  - wild 265
- portability 209

- porting 123, 176
- ppppppppp 198
- pre-defined macros
  - \_\_INSURE\_\_ 97, 107
- pre-processor symbols 97
- printf 214–218
- printf 111, 125, 129, 356
- printf checking 48
- process ID 70
- process ID (%p), in error report banners 184
- process ID (%p), in filenames 178
- production code 126
- prototypes 49
- prototypes, function, used as interfaces 119
- .psrc options
  - interface\_library 122
  - inuse 136, 140
  - rename\_files 99
- .psrc options
  - interface\_library 123
  - suppress 134

## Q

- qsort 130
- Quality Consulting 15
- Query 143
  - editor 152
  - evaluating 154

## R

- %R, filename macro 177
- %r, filename macro 177
- read, READ\_UNINIT\_MEM sub-category 45
- READ\_OVERFLOW 36
  - overflow diagrams 197
- READ\_UNINIT\_MEM
  - copy sub-category 45
  - read sub-category 45
- readdngl.c 303

- readindx.c 301
  - readnull.c 305
  - readover.c 311
  - readovr1.c 307
  - readovr2.c 45, 308
  - readovr3.c 309
  - readuni1.c 45, 313
  - readuni2.c 314
  - readuptr.c 317
  - readwld1.c 319
  - readwld2.c 320
  - realloc 44
  - red, color in Inuse 144
  - rename\_files 99
  - repeated errors 70
  - replacing malloc 185
  - report summaries 77
  - report\_file, Advanced Options 68
  - reports
    - default behavior 67
    - filename generation 68
  - retdngl.c 322
  - retfail.c 324
  - retinc.c 326
  - <return> 199
  - return values
    - checking automatically 51
    - ignoring 41
  - RETURN\_FAILURE 44, 51–52, 245, 293
  - RETURN\_FAILURE 125, 356
  - rrrrrrrrrr 198
  - Running Insure++
    - with CodeWizard 32
    - with Inuse 34
    - with TCA 33
  - Running Inuse 140–154
  - running out of memory 42
  - Running TCA 157–173
- S**
- scandir 130
  - scanf 214–218
  - scanf 355
  - scanf checking 48
  - search for source code 72
  - shared memory 37, 52
  - sign (error sub-category) 214, 221
  - signal 130
  - signal handlers 107, 130
  - signal\_catch, Advanced Options 108
  - signal\_ignore, Advanced Options 108
  - Signals 107–108
  - 16-bit machines 47
  - 64-bit machines 47
  - sizeof operator 183
  - slowleak.c 135, 136, 137
  - source directories 72
  - source\_path, Advanced Options 72
  - sprintf, see printf
  - sqrt 331
  - sscanf, see scanf
  - stack trace 67
  - stack\_internal, Advanced Options 104, 106
  - states
    - chaperon 56
  - static variables 37
  - stderr 67
  - stretch1.c 76
  - stretch2.c 77
  - stretchy arrays 76, 179, 184
  - strings
    - declaring in interfaces 353
    - errors using 44, 309
  - strncpy 45, 308
  - structure, variable length 76, 179
  - structures, anonymous 77
  - suffixes, see file extensions
  - summaries 77
  - Summarize Leaks 41
  - suppress 134
  - suppress\_output, Advanced options 75
  - suppressing
    - C++ warnings 75
    - error messages 73, 89
    - warnings 75

## Index

system calls 51, 52  
system name 70

## T

`%T`, filename macro 177  
`%t`, filename macro 177  
`%t`, time macro 184  
TCA 33  
    coverage analysis 163  
    display 169  
    introduction 155  
    loading a report file 170  
    options 196  
    `tca.log` 163, 164, 167, 195  
    test coverage data 160  
    using 157–173  
`tca.log` 163, 164, 167, 195  
technical support 15  
32-bit machines 47  
time (`%t`), in error report banners 184  
time and date, on error reports 70  
Time layout 143, 148  
    `.tqs` file extension 122  
    `.tqs` file extension 111  
trace, Advanced Options 104, 106  
trace.C 106  
trace\_banner, Advanced Options 105  
trace\_file, Advanced Options 105  
tracing 104–106  
    output to a file 105  
    turning on 104  
    typical output 104  
tracing, Advanced Options 104–106  
type promotion 183  
type-checking, via interfaces 119

## U

uninitialized memory 45  
union (error sub-category) 221  
unions, anonymous 77  
unrepeatable errors 45

unsuppress, Advanced Options 76  
unuassign.c 328  
unused variables 46  
unuvar.c 329  
Usage Summary 143, 149  
USER\_ERROR 356  
usererr.c 331  
using interfaces 110

## V

`%V`, filename macro 178  
`%v`, filename macro 178  
variable arguments 129, 356, 357  
variable declarations  
    incompatible 47  
variable length structures 76, 179  
variables  
    uninitialized 45  
    unused 46  
vfprintf 356  
Viewing Source Files 92  
virtbad1.C 333  
virtbad2.C 334  
virtbad3.C 334  
void `_Insure_trace_annotate` 105

## W

warn.c 276  
warnings  
    compile time 50  
    suppressing 75  
wild pointers 265  
writdngl.c 338  
WRITE\_OVERFLOW 36, 197  
writindx.c 336  
writing interfaces 117  
writnull.c 340  
writover.c 342  
writuptr.c 344  
writwld1.c 346  
writwld2.c 347

wwwwww 197

## X

\x escape sequence 182

X Window System 130

XtAddCallback 132

## Y

yellow, color in Inuse 144

Index

Index