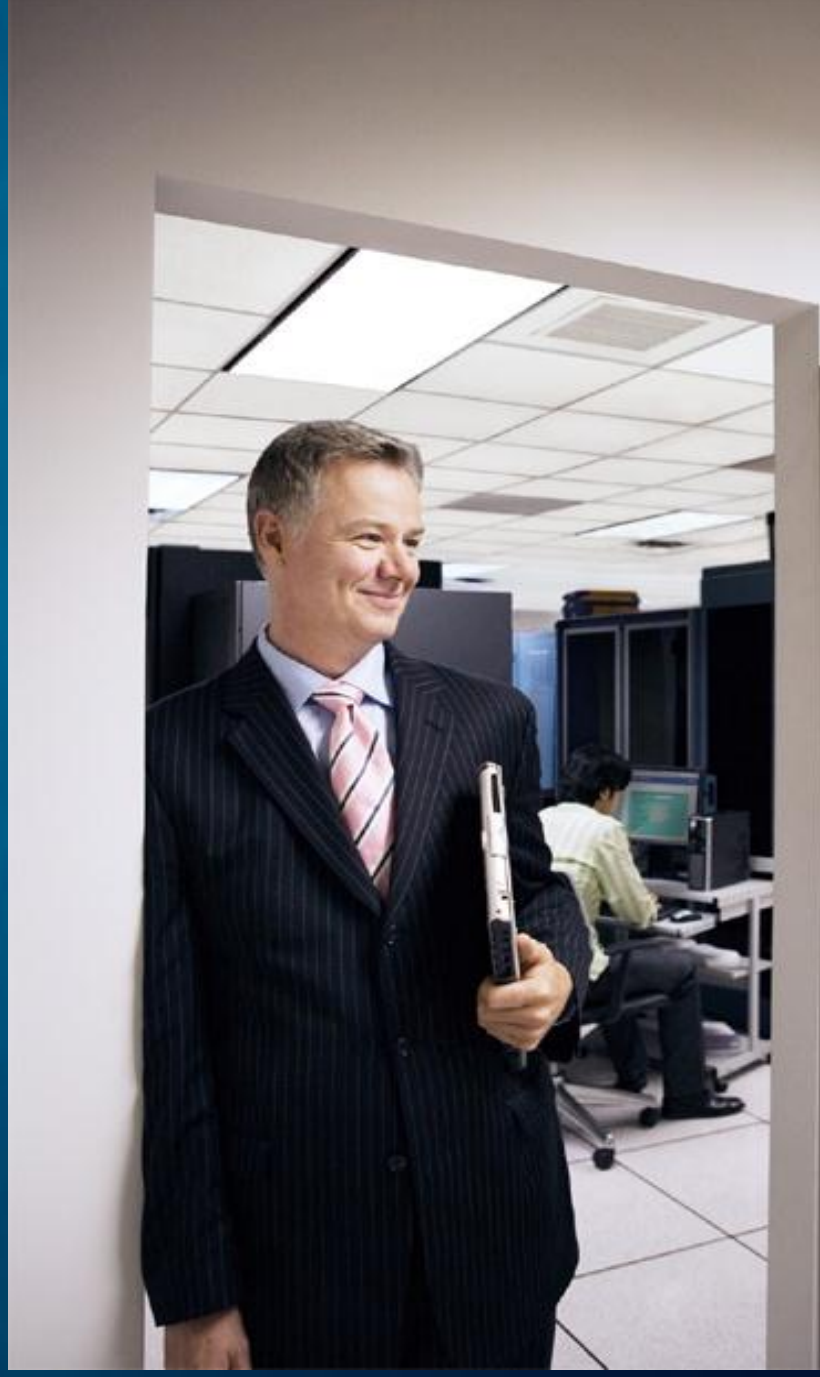
Pointer Checker Feature in Intel® C++ Composer 2013: Catch Out-of-Bounds Memory Accesses easily!

Intel Technical Presentation
May 22, 2012

Kittur Ganesh,
Technical Consulting Engineer,
Intel® Software Development Products



Agenda

- What is Pointer Checker?
- Pointer Checker Enabling – *At a glance*
- Pointer Checker Usage Model
- Using Pointer Checker
  - *Checking Bounds*
  - *Checking for Dangling Pointers*
  - *Checking Arrays*
  - *Intrinsics*
  - *Working with Enabled and Non-Enabled Modules*
  - *Checking Run Time Library (RTL) functions*
  - *Finding and Reporting Out-of-Bounds Errors*
  - *Guidelines*
- Pointer Checker – *Performance Overhead*
- Summary

What is Pointer Checker?

- C/C++ pointers have well defined semantics that determine range of memory to access.
  - Compilers typically do not enforce

```
p = malloc(size);
• lower_bound(p) is (char *)p
• upper_bound(p) is lower_bound(p) + size - 1
```

- Buffer Overflow/Overrun anomaly
  - Violation of memory safety
  - Data corruption
  - Erratic program behavior
  - Breach of system security
  - Basis of many software vulnerabilities

```
char *buf = malloc(5);
for (int i=0; i<=5;i++) {
      buf[i] = 'A' + i;
}
```

What is Pointer Checker?

Pointer Checker
What is it?

- A key feature of Intel® C++ Composer XE 2013
- Designed for use during application debugging and testing
- Enabled via compile time switches.
- User API allows control over what happens when a violation is detected
- Implemented mostly in a runtime library which is automatically linked in by the compiler
- No change to structure layout or ABIs

Pointer Checker
Key Benefits

- Catches out-of-bounds memory accesses
  - Identifies and reports before memory corruption occurs!
- Finds memory buffer overruns
  - Checks memory accesses through pointers
  - Includes subscripted array accesses
- Finds dangling pointers
  - Checks memory accesses through freed pointers

<u>Key Benefit: Enable Incrementally</u>
Pointer Checker can be enabled on a single file, group of files or all files. Pointer Checker enabled code and non-enabled code can coexist!

Pointer Checker – *How it works?*
High Level Design

- The compiler:
  - Creates bounds when a pointer is created via the "&" operator or array reference.
  - Copies bounds when a pointer is copied.
  - Stores bounds when a pointer is stored in memory.
  - Loads bounds when a pointer is loaded from memory.
  - Passes bounds with pointer arguments and function returns.
  - Generates checks when a pointer is used for indirect memory references.
- Runtime library wrappers:
  - Create bounds when memory is allocated
  - Bounds follows the pointer. Casting doesn't change the bounds of the pointer.
  - Checks bounds for pointer parameters (Example: strcpy())

Pointer Checker Enabling - *At a glance*
Getting started is easy…

- Meet requirements:

Supported Languages	Supported Architecture	Supported Platforms	Supported Processor
C, C++	IA-32, Intel® 64	Linux*, Windows*	Intel® Pentium® 4 processor or later, or compatible non-Intel processor

- Compile and build your application with:

  -check-pointers=[none | write | rw] (Linux* OS)

  /Qcheck-pointers:[none | write | rw] (Windows* OS)

  – Pointer Checker is off by default
  – Checks all indirect accesses through pointers and accesses to arrays.

One compiler switch enables Pointer Checker!

Pointer Checker Enabling - *At a glance*
A sample program

- Compile with Pointer Checker enabling option
- Execute;Check for out-of-bounds errors (OOB)
- Determine if OOB is true or false positive

```
#include<stdio.h>
#include<chkp.h>

int main () {

#ifdef REPORT
  __chkp_report_control(__CHKP_REPORT_TRACEBACK, 0);
#endif

    char *my_chptr = "abc";
    char *another_chptr;
    another_chptr = (char *) malloc (strlen(( char *)my_chptr));
    printf ("sizeof another_chptr  is %d\n", strlen(( char
*)my_chptr));
    printf ("sizeof my_chptr is %d\n", sizeof(my_chptr));
    memset (another_chptr, '@', sizeof(my_chptr));   /* Line 15 */
    printf ("after memset = %s\n", another_chptr);
    return 0;
}
```

- Compile without Pointer Checker enabling switch:

```
% icc main.c;./a.out
sizeof another_chptr  is 3
sizeof my_chptr is 8
after memset = @@@@@@@@
```

- Compile with Pointer Checker enabling option:

```
% icc main.c -DREPORT -check-pointers=write -rdynamic -
g;./a.out
sizeof another_chptr  is 3
sizeof my_chptr is 8
CHKP: Bounds check error
Traceback is:
./a.out(__chkp_check_bounds+0x1f1) [0x403a31]
./a.out(__chkp_memset+0x68) [0x404078]
./a.out(main+0x334) [0x4032b8]
/lib64/libc.so.6(__libc_start_main+0xfd) [0x7fba1b43ebfd]
./a.out() [0x402ec9]
%
```

- Map address to source line where OOB occurs

```
% addr2line -e ./a.out 0x4032b8
main.c:15
%
```

- CHKP_REPORT_TRACEBACK?

Pointer Checker – *Usage Model*

Model	Description	
Header File	Defines intrinsics and reporting functions (chkp.h)	
Compiler Options	-check-pointers (/Qcheck-pointers)	Enables pointer checker and adds associated libraries
	-check-pointers-dangling (/Q-check-pointers-dangling)	Enables checking for dangling pointer references
	-check-pointers-undimensioned (Qcheck-pointers-undimensioned)	Enables the checking of bounds for arrays without dimensions
Intrinsics	void * __chkp_lower_bound(void **)	Returns the lower bound associated with the pointer
	void * __chkp_upper_bound(void **)	Returns the upper bound associated with the pointer
	void * __chkp_kill_bounds(void *p)	Removes the bounds information to allow the pointer in the argument to access all memory.
	void * __chkp_make_bounds(void *p, size_t size)	Creates new bounds information within the allocated memory address for the pointer in the argument
Reporting API (Function/Enumeration)	void __chkp_report_control(__chkp_report_option_t option, __chkp_callback_t callback)	Determines how errors are reported
	__chkp_report_option_t {Enumerations: __CHKP_REPORT_LOG, __CHKP_REPORT_TRACEBACK, __CHKP_REPORT_CALLBACK, __CHKP_REPORT_BPT, __CHKP_REPORT_TERM}	Controls how out-of-bounds error are reported. Enumerations in header file
RTL Functions	Provides checking on C run-time library functions that manipulate memory through pointers	

Using Pointer Checker - *Checking Bounds*

Checks indirect accesses through pointers for accesses that are out of bounds

- Check Bounds on Read/Write Operations

  -check-pointers=[none | write | rw] (Linux* OS)

  /Qcheck-pointers:[none | write | rw] (Windows* OS)

```
%cat main.c
#include<stdio.h>
#include<malloc.h>
#include <chkp.h>

int main () {
#ifdef REPORT
  __chkp_report_control(__CHKP_REPORT_TRACEBACK, 0);
#endif
  char *buf = malloc(4);
  int i;
  for (i=0; i<=4; i++) {
     printf(" %c",buf[i]); /* Line# 12 */
  }
  for (i=0; i<=4; i++) {
     buf[i] =  'A' + i;      /* Line# 15 */
     printf(" %c",buf[i]);
  }
  printf ("\n");
  return 0;
}
```

- Compile without Pointer Checker enabling switch:

```
% icc main.c -g;./a.out
A B C D E
```

- Compile with Pointer Checker enabling option:

```
% icc main.c -DREPORT -check-pointers=write -rdynamic -
g;./a.out
CHKP: Bounds check error
Traceback is:
./a.out(__chkp_check_bounds+0x1f1) [0x4033c1]
./a.out(main+0x21f) [0x402c83]
/lib64/libc.so.6(__libc_start_main+0xfd) [0x3b7d41ec5d]
./a.out() [0x4029a9]
```

- Map address to source line where OOB occurs

```
% addr2line -e ./a.out 0x402c83
main.c:15
%
```

- An out-of-bounds error was not reported for line#12. Why?

Using Pointer Checker - *Checking For Dangling Pointers*

- Check for dangling pointers in stack and heap

  -check-pointers-dangling=[none | heap | stack | all] (Linux* OS)

  /Qcheck-pointers-dangling:[none | heap | stack | all] (Windows* OS)

- When enabled:
  - Compiler uses a wrapper for the C runtime function free() and the C++ delete operator.
  - Compiler sets dangling pointer bounds to: lower_bound(dp) = 2; upper_bound(dp)=0;

    Why are the bounds set as above?

```
%cat dang.c
#include<stdio.h>
#include <malloc.h>
#include <chkp.h>

char * test() {
  char *dp = malloc(6);
  strcpy(dp, "hello");
  free(dp);
  return dp;   /* dp is dangling pointer now */
}

int main () {
#ifdef REPORT
  __chkp_report_control(__CHKP_REPORT_TRACEBACK, 0);
#endif
  char *q = test();
  printf("first ch = %c\n", *q); /* Line 17 */
}
```

% icc dang.c -DREPORT -check-pointers=rw -check-pointers-dangling=all -rdynamic -g;./a.out
CHKP: Bounds check error
Traceback is:
./a.out(__chkp_check_bounds+0x1f1) [0x403a11]
./a.out(main+0x105) [**0x40332b**]
/lib64/libc.so.6(__libc_start_main+0xfd) [0x7f9640479bfd]
./a.out() [0x402ef9]

- Map address to source line where OOB occurs

%addr2line –e ./a.out **0x40332b**
dang.c:17

- Uncheck check-pointers-dangling

% icc -DREPORT -check-pointers=rw -check-pointers-dangling=none -g dang.c;./a.out
first ch =
%

Using Pointer Checker - *Checking Arrays*

- Pointer checker checks arrays in modules that actually define the arrays with bounds
- For checking of bounds for arrays without dimensions:
  - -[no-]check-pointers-undimensioned (Linux* OS)
  - /Qcheck-pointers-undimensioned[-] (Windows* OS)

```
%cat arr.c
#include<stdio.h>
#include <chkp.h>

extern int A[];
int main () {
#ifdef REPORT
  __chkp_report_control(__CHKP_REPORT_TRACEBACK, 0);
#endif
  A[3] = 1;
  A[5] = 2; /* OOB Line 10 */
  return 0;
}

%cat arr1.c
int A[5]
```

```
% icc -DREPORT -check-pointers=write -check-pointers-
undimensioned -rdynamic -g arr.c arr1.c; ./a.out
CHKP: Bounds check error
Traceback is:
./a.out(__chkp_check_bounds+0x1f1) [0x401d71]
./a.out(main+0x88) [0x4016ec]
/lib64/libc.so.6(__libc_start_main+0xfd)
[0x7fbaa24bebfd]
./a.out() [0x4015a9]
```

- Map address to source line where OOB occurs

```
% addr2line -e ./a.out 0x4016ec
arr.c:10
%icc -DREPORT -check-pointers=write -no-check-
pointers-undimensioned -rdynamic -g arr.c arr1.c;
./a.out
%
```

Using Pointer Checker - *Intrinsics*

- Defined in header file <chkp.h>
- Ideal for:
  - Writing your own wrappers for Run-Time Library (RTL) functions
  - Working with enabled and non-enabled Modules
  - Checking and creating correct bounds for Custom Memory Allocators
  - &c

Intrinsics:

void * __chkp_kill_bounds(void *p)

➢ Kills the descriptor associated with the pointer making all memory accessible via the returned pointer.

Void * __chkp_make_bounds(void *p, size_t size)

➢ Make bounds for a pointer. The lower bounds is pointer, and the upper bound is pointer + (size – 1).

void * __chkp_lower_bound(void **) / void * __chkp_upper_bound(void **)

➢ Retrieves the lower / upper bound associated with a pointer

For example (setting exact bounds)

```
void *myalloc(size_t size) {
    // Code allocating the large chunk of memory
    // into small chunks.
    // Add bounds information to the pointer
    return __chkp_make_bounds(p, size);
 }
```

For example: An Allocation wrapper

```
extern void *wrap_malloc(size_t bytes) {
    void* p; p= malloc(bytes);
    if (p) {
       p = (void*)__chkp_make_bounds(p,bytes);
    }
    return p;
}
```

Using Pointer Checker – *Working with Enabled and Non-Enabled Modules*

- With non-enabled code writes or returns, false positives occur in enabled code, as bounds aren't set correctly
  - Pointer Checker mitigates this for nearly all cases by checking the stored pointer against a copy of the pointer stored with the bounds.
  - Pointers can still match in some cases such as realloc() returning same pointer but larger object:
    - p = my_realloc(p, old_size + 100);

- Solution:
  - Use wrapper functions when calling non-Pointer Checker code that kills or sets the bounds correctly for any pointer returned or written by the function.

Using Pointer Checker – *Checking RTL Functions*

- Run-Time Library (RTL) routines dealing with pointers need to be encapsulated or replaced so returned pointers have proper descriptors, and usage of pointers within the RTL routine are checked correctly.

- Pointer Checker provides checking on RTL functions which manipulate memory through pointers
  - Uses library of functions or wrappers
  - Pointer Checker Wrapper Library: <libchkpwrap.a> [Linux*]
    <libchkpwrap.lib> [Windows*]
- To find which run-time routines are wrapped:
  - Example (Linux*): %nm libchkpwrap.a | egrep 'T __chkp_`
  - The returned list signify wrappers such as:
    __chkp_strcpy - the wrapper for strcpy()

Using Pointer Checker – *Finding and Reporting Out-of-Bounds Errors*

- Reporting controlled through:
  - __chkp_report_option_t enumeration
  - __chkp_report_control() library function

Enum Value	Action
__CHKP_REPORT_NONE	Do nothing.
__CHKP_REPORT_BPT	Execute a breakpoint interrupt.
__CHKP_REPORT_LOG	Log the error and continue; the compiler will report each out-of-bounds pointer it finds.
__CHKP_REPORT_TERM	Log the error and exit the program; the compiler will only report the first bounds violation and then terminate.
__CHKP_REPORT_CALLBACK	Call a user defined function; the compiler will invoke a user-defined function to deal with a bounds error.
__CHKP_REPORT_TRACEBACK	Report stack traceback, including instruction addresses. This is the default report mode. On Windows* OS, specify the /Zi compiler option to get better traceback information, including routine names. On Linux* OS, specify the -rdynamic link command.

- For example, to report all bounds errors:
  - __chkp_report_control(__CHKP_REPORT_LOG, 0);

Using Pointer Checker – *Guidelines*

- Use Debug Configuration when using Pointer Checker for testing and debugging, so symbols are visible for better trace-back functionality. Use –rdynamic linker option when compiling on Linux*.
- Compile to make bounds error occur near bad pointer generation. Compile with:
  - No optimization (avoids optimizing out memory accesses and improves source line correlation)
  - Check both READS and WRITES to reduce fault delay.
- Use __CHKP_REPORT_LOG option to analyze loop issues in conjunction with __CHKP_REPORT_TRACEBACK
- Release application with Pointer Checker disabled:
  - Application size and execution time increases with Pointer Checker enabled.

Pointer Checker - *Performance Overhead*

- Runtime cost is high, about 2X to 5X execution time effect.
- Code size increase from 20% to a very large increase (>100% plus), depending on the application.
- Pointer Checker is seen as a debug tool.
- Deployed applications are expected to have Pointer Checker disabled
- Security benefits from catching vulnerabilities prior to product release is the trade-off.

Summary

- Pointer Checker is a key feature of Intel® C++ Composer XE 2013.
- Pointer Checker is designed for use during application debugging and testing.
- Point Checker provides full checking of all memory accesses through pointers.
- A Pointer Checker enabled application will catch any out-of-bounds memory accesses before any memory corruption occurs.
- Pointer Checker enabled code and non-enabled code can coexist.
- Get Started with Pointer Checker!

Optimization Notice

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Legal Disclaimer

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference www.intel.com/software/products.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino Atom, Centrino Atom Inside, Centrino Inside, Centrino logo, Cilk, Core Inside, FlashFile, i960, InstantIP, Intel, the Intel logo, Intel386, Intel486, IntelDX2, IntelDX4, IntelSX2, Intel Atom, Intel Atom Inside, Intel Core, Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, Itanium, Itanium Inside, MCS, MMX, Oplus, OverDrive, PDCharm, Pentium, Pentium Inside, skoool, Sound Mark, The Journey Inside, Viiv Inside, vPro Inside, VTune, Xeon, and Xeon Inside are trademarks of Intel Corporation in the U.S. and other countries.

*Other names and brands may be claimed as the property of others.

http://intel.com/software/products

Resources & Links

- Intel® Compilers product website:
  http://www.intel.com/software/products/compilers

- Active User Forum:
  http://software.intel.com/en-us/forums/

- Software Development Products Knowledge Base
  http://software.intel.com/en-us/articles/tools/

- To evaluate Intel® Software Development Products for your High Performance needs:
  http://www.intel.com/software/products/eval