What is Valgrind?

- A Suite of Free and Open Source Debugging and Profiling tools
- Can detect many memory-related errors commonly found in C and C++
- Used industry-wide.
- Memcheck
  - Most popular of these tools

Quick Start Guide

- Compile your code in debug mode
  - Give –g to gcc/g++
  - You may also use –o1
    - Line numbers in the message may be incorrect
  - Do not use –o2 or anything above
- Command to run

```
valgrind --leak-check=yes prog <args>
```

- Few finer points
  - Memcheck is the default tool. To use other tool, you give

```
valgrind –tool=callgrind prog <args>
```

Let’s debug some memory error

```
#include <iostream>

using namespace std;

void func1() {
   int* vec = new int [10];
   for (int i = 0; i <= 10; ++i) {
      vec[i] = 1;
   }
}

int main() {
   func1();
   return 0;
}
```

```
==11888== Invalid write of size 4
==11888==    at 0x4006F0: func1() (main1.cxx:8)
==11888==    by 0x40070F: main (main1.cxx:13)
==11888==  Address 0x4c36068 is 0 bytes after a block of size 40 alloc'd
==11888==    at 0x4A0674C: operator new[](unsigned long)
(vg_replace_malloc.c:305)
==11888==    by 0x4006D5: func1() (main1.cxx:6)
==11888==    by 0x40070F: main (main1.cxx:13)
==11888==
==11888==
==11888== HEAP SUMMARY:
==11888==     in use at exit: 40 bytes in 1 blocks
==11888==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==11888==
==11888== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==11888==    at 0x4A0674C: operator new[](unsigned long)
(vg_replace_malloc.c:305)
==11888==    by 0x4006D5: func1() (main1.cxx:6)
==11888==    by 0x40070F: main (main1.cxx:13)
==11888==
==11888== LEAK SUMMARY:
==11888==    definitely lost: 40 bytes in 1 blocks
==11888==    indirectly lost: 0 bytes in 0 blocks
==11888==      possibly lost: 0 bytes in 0 blocks
==11888==    still reachable: 0 bytes in 0 blocks
==11888==         suppressed: 0 bytes in 0 blocks
==11888==
==11888== For counts of detected and suppressed errors, rerun with: -v
==11888== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 6 from 6)
```

Understanding the output of Memcheck

- ==29403== Invalid write of size 4
  - Process id : 29403
  - Type of error : “Invalid write”
  - Below it, the stack trace.
- ==11888== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
  - Memory leak followed by the stack trace
- Note: Valgrind doesn’t detect static buffer overflow

```
int vec2[10];

vec2[10] = 4;
vec2[11] = 5;
```

Memory Leaks

- What is a memory leak?
  - There's a memory chunk allocated, but you can't access that
- Memcheck reports following 4 kinds of leaks
  1. Still Reachable
     - Still have pointer(s) to the start of the block
     - No problem. You can still free it. By default not reported
  2. Definitely Lost
     - No pointer to the memory block could be found
     - Can not be freed at the end of the program
  3. Indirectly Lost
     - All the pointers that point to the block are lost
     - For instance if root node of the binary tree is lost, all its children are indirectly lost
  4. Possibly Lost
     - There's a chain of one or more pointers, but one of the pointers is an interior pointers

Memory Error

- Different types of errors
  - Illegal read / Illegal write error
    - This happens when your program reads or writes memory at a place which Memcheck reckons it shouldn't

```
void func1() {
   int* vec = new int [11];
   for (int i = 0; i <= 10; ++i) {
      vec[i] = 1;
   }
   delete vec;

   vec[10] = 4;
}
```

Memory Error (Contd.)

- Different types of errors
  - Use of Uninitialized values
    - when your program uses a value which hasn't been initialised -- in other words, is undefined
    - Sources of uninitialized value
      - Local variables in procedures which have not been initialised
      - The contents of heap blocks before you write something there

```
void func1() {
   int* vec = new int [11];
   if (vec[0] == 4) {
      cout << "Initialized" << endl;
   } else {
      cout << "Not initialized" << endl;
   }
}
```

Memory Error (Contd.)

- Different types of errors
  - Use of uninitialized values in system calls

```
int main( void ) {
 char* arr = malloc(10);
 int* arr2 = malloc(sizeof(int));
 write( 1 /* stdout */, arr, 10 );
 exit(arr2[0]);
 }
```

1. Syscall param write(buf) points to uninitialised byte(s)
2. Syscall param exit(error_code) contains uninitialised byte(s)

Memory Error (Contd.)

- Different types of errors
  - Illegal Free
    - Program freeing memory block twice
  - Heap block freed with inappropriate deallocation function
    - If allocated with malloc, calloc, realloc, valloc or memalign, you must deallocate with free.
    - If allocated with new, you must deallocate with delete.
    - If allocated with new[], you must deallocate with delete[].
  - Overlapping source and destination
    - Probable places – memcpy, strcpy etc.

Some useful options of Memcheck

```
--num-callers=<number> [default : 12]
```

- Maximum number of entries shown in the stack trace

```
--log-file=<file name>
```

- If you use –log-file=<file name>.%p, then the process ID will be added. For instance

```
$ valgrind --log-file=valgrind.log.%p ./main1
$ ls
$ main1   main1.cxx   valgrind.log.2866
```

```
--trace-children=<yes|no> [default: no]
```

- Needed if your program creates sub-processes through exec system call.

- valgrind –h will list you many other basic user options

Some useful options of Memcheck

- A very common message seen is – "Conditional jump or move depends on uninitialised value(s)“
  - Memcheck reports use of uninitialised values.
  - To know sources of uninitialised data, use the following option

```
--track-origin=yes [default : no]
```

- Suppressing errors
  - You may need it to suppress errors in library code.

```
--gen-suppressions=yes
```

  - Valgrind will pause after every error and ask you to print the suppression. Press Y.
  - Copy all the messages into a file (e.g. my.supp) and in the future valgrind run, use

```
valgrind --leak-check=yes --suppressions=./my.supp ./main1
```

Suppressing errors

```
==14682== Conditional jump or move depends on uninitialised value(s)
==14682==    at 0x400873: func1() (main1.cxx:7)
==14682==    by 0x4008B9: main (main1.cxx:15)
==14682==
==14682==
==14682== ---- Print suppression ? --- [Return/N/n/Y/y/C/c] ---- y
{
  <insert_a_suppression_name_here>
  Memcheck:Cond
  fun:_Z5func1v
  fun:main
}
Not initialized
==14682==
```

Passing options to Valgrind

- Valgrind read options from following 3 places
  - ~/.valgrindrc
  - $VALGRIND_OPTS
  - ./.valgrindrc

Memcheck - overview

- Detects the following errors
  - Accessing memory you shouldn't
    - But it can't detect memory errors on statically allocated memory.
  - Using undefined values
  - Incorrect freeing of heap memory
  - Memory leaks etc.
- Usually, your program should have no error.
- GUI for Valgrind
  - Valkyrie
  - Alleyoop
  - MemcheckView

How does Valgrind(Memcheck) Work?

- Memcheck adds instrumentation code to the executable
- Valgrind core coordinates the execution of the instrumented code
- V-Bits
  - Every bit of data in your program is associated with a “Valid Value Bit” or V-Bit
  - Establishes validity of value
  - Copying values around does not cause Memcheck to check for, or report on, errors.
  - However, when a value is used in a way which might conceivably affect your program's externally-visible behaviour, the associated V bits are immediately checked.

How does Valgrind(Memcheck) Work?

- A-Bits
  - To check if the data at that location should be accessed or not
  - Before every read/write Memcheck checks A-Bits
  - Each byte has a single A-bit

List of Other Valgrind Tools

- Cachegrind
  - Cache profiler. Pinpoint source of cache-miss
- Callgrind
  - Provides call graph along with above output
- Helgrind
  - Thread debugger. Finds data races etc
- Drd
  - Detects errors in multithread C/C++ programs
- Massif
  - Heap profiler – heap usage etc

- etc