MAE 5032 High Performance Computing: Methods and Practices

Lecture 8: Defensive programming and debugging

Ju Liu

Department of Mechanics and Aerospace Engineering

liuj36@sustech.edu.cn

Defensive Programming

Codes and bugs

- Developing defensive programming, debugging, and profiling skills:
  - You introduce bugs at some point in your code
  - Even if you run code from libraries, they may also introduce bugs
  - Commerical applications have bugs too

- Bugs have huge impact on our society
  - In 1962, the Mariner-1 space probe mission failed due to a missing hyphen in the source code.

Codes and bugs

- Developing defensive programming, debugging, and profiling skills:
  - You introduce bugs at some point in your code
  - Even if you run code from libraries, they may also introduce bugs
  - Commerical applications have bugs too

- Bugs have huge impact on our society
  - In 2009, a bug in the anti-lock-break software installed in Toyota Lexus resulted in a recall of 9 million cars and the death of 4 people.

'There's no brakes... hold on and pray': Last words of man before he and his family died in Toyota Lexus crash.

Codes and bugs

- Developing defensive programming, debugging, and *profiling* skills:
  - You introduce bugs at some point in your code
  - Even if you run code from libraries, they may also introduce bugs
  - Commerical applications have bugs too

- Bugs have huge impact on our society

- Bugs cost time: an average developer spends 50% of his/her time debugging

- Bugs impact science:

  A Nature news features discusses leaked emails on climate research in which one of the researchers repeatedly refers to problems in his software as “Yup, my awful programming strikes again.” The article calls for professional training for scientists who develop software packages.

Taxonomy of Bugs

High-level taxonomy

- Structural bugs
- Arithmetic bugs
- Data race and deadlocks
- Bugs in data

Structural bugs

- This is a very broad category that can be divided into a number of subcategories
  - control flow: loop termination
  - logic
  - processing
  - initialization

```
const int n = 5;
double data[n];
for (int i = 0; i <= n; i++)
    data[i] = some_function(i);
```

We need to switch between 0-based language and 1-based language oftentimes.
It can be detected by compiler or valgrind.

Structural bugs

- This is a very broad category that can be divided into a number of subcategories
  - control flow
  - logic
  - processing
  - initialization

We need proper testing to detect this type of bug.

```
subroutine print_classification(x)
    implicit none
    real, intent(in) :: x
    real, parameter :: low = -5.0, high = 5.0
    if (x < low) then
        print '(A)', 'low'
    else if (low < x .and. x < high) then
        print '(A)', 'medium'
    else
        print '(A)', 'high'
    end if
end subroutine print_classification
```

Structural bugs

- This is a very broad category that can be divided into a number of subcategories
  - control flow
  - logic
  - processing
  - initialization

```
char *text;
...
if (strlen(text) && text != NULL)
    process(&text);
```

```
char *text;
...
if (text != NULL && strlen(text))
    process(&text);
```

Which one is better?

Structural bugs

- This is a very broad category that can be divided into a number of subcategories
  - control flow
  - logic
  - processing
    - memory leaks
    - open a file but not close it in I/O
    - MPI communications require a pair of function calls to start and end the communication.
  - initialization

Structural bugs

- This is a very broad category that can be divided into a number of subcategories
  - control flow
  - logic
  - processing
  - initialization

```
program unallocated
    use, intrinsic :: iso_fortran_env, only : error_unit
    implicit none
    integer :: n, i
    real, dimension(:), allocatable :: data
    real :: r
    character(len=80) :: buffer

    if (command_argument_count() < 1) then
        write (fmt='(A)', unit=error_unit) &
            'missing command argument, positive integer expected'
        stop 1
    end if
    call get_command_argument(1, buffer)
    read (buffer, fmt='(I10)') n
    do i = 1, n
        call random_number(r)
        data(i) = r
    end do
    print '(A, F15.5)', sum(data)
end program unallocated
```

The variable data has been declared to be **allocatable**, but the allocate statement is missing.
Often can be found by compiler or valgrind.

Arithmetic bugs

- Integers typically are held by 32 bits.

- Overflow: the largest signed integer is 2^{31}-1 and the smallest is -2^{31}. Beyond this range, the code will have an overflow bug.
  - This bug can be trapped by compilers.

- Divide by zero: an integer division by zero will result in runtime error. The application will crash.

Arithmetic bugs

- Real numbers typically are held by 32 bits (single precision), 64 bits (double precision), or 128 bits (quadruple precision).
- Overflow: a result larger than the largest floating point number will be Infinity and further calculations will result in either Infinity or NaN (Not a Number).
  - This bug can be trapped by compilers.
- Underflow: the smallest strictly positive floating point number that can be represented is $1.17549435 \times 10^{-38}$. Result smaller than it will be rounded to zero, which is an underflow.

Data race and deadlocks

- A data race will occur when two or more threads
  - access the same memory location concurrently;
  - at least one thread accesses that memory location for writing;
  - no implicit or explicit locks are used to control access.

- A deadlock occurs in a concurrent system when each process is waiting for some other process to take action.
  - Ex: Circular wait: each process must be waiting for a resource which is being held by another process, which in turn is waiting for the first process to release the resource. In general, there is a set of waiting processes, P = {P1, P2, ..., PN}, such that P1 is waiting for a resource held by P2, P2 is waiting for a resource held by P3 and so on until PN is waiting for a resource held by P1.

Bugs in data

- A fairly large number of bugs is caused by inappropriate data conversion.

```c
#include <stdio.h>

int main() {
    long a = 94850485030;
    long b = 495849853000;
    int c = a + b;
    printf("c = %d\n", c);
    double x = 1.435e67;
    double y = 4.394e89;
    float z = x + y;
    printf("z = %e\n", z);
    int d = x;
    printf("d = %d\n", d);
    return 0;
}
```

Defensive programming tips

- “Code formatting is about communication, and communication is the professional developer’s first order of business.”
  - ➢ use indentation and spacing
  - ➢ https://google.github.io/styleguide/cppguide.html

- Use language idioms

```
int factorial(int n) {
    fac = 1;
    for (int i = 2; i <= n; i++)
        fac = fac*i;
    return fac;
}
```

fac *= i

Use language idioms

- “Code formatting is about communication, and communication is the professional developer’s first order of business.”

- Use language idioms

```
REAL, DIMENSION(10) :: a
...
a = value
```

```
INTEGER :: i
REAL, DIMENSION(10) :: a
...
DO i = 1, 10
    a(i) = value
END DO
```

Descriptive names

- Variable names should be nouns and function names should be verbs.

- Limit the scope of variables. In old days, you have to declare all variables at the start of a block. Now, modern languages allow you to declare variable anywhere before their first use. Limiting the scope of declaration to a minimum reduces the probability of inadvertently using the variable.

- Be explicit about constants.

Descriptive names

- Control the access of object attributes.

access modifier	C++	Fortran
private	access restricted to class/struct	access restricted to module
protected	access restricted to class/struct and derived	variables: modify access restricted to module, read everywhere
public	attributes and methods can be accessed from everywhere	variables, types and procedures can be accessed from everywhere
none	class: private, struct: public	public

- Initialize values of variables.

- Comment your code and make it up-to-date.

Check errors

- Checking errors is not a waste of time, and it actually may save your long debugging sessions.
- Example: make sure the memory is allocated successfully:

```
 p = (float
*)calloc(nnodes+1,sizeof(float));
 if(p == NULL)
   {
     printf("Allocation error for p!\n");
     exit(1);
    }
```

Initialization and Compilers

- Be sure to initialize all variables and arrays that require it (don’t count on the artchitecture/OS to do this for you)

- During the testing and validation phase, use of available compiler options
  - -Wall –Wextra –Wshadown –Wunreachable-code –Wuninitialized
  - Intel Fortram examples
    - -check all : enable runtime checks for out-of-bounds array subscripts, uninitialized variables, etc.
    - -warn all : display all relevant warning messages
    - -warn errors : tells the compiler to change all warning level messages into error-level messages
    - -fpe0 : tells the compiler to abort when any flowing point exceptions occur.

Consult your compiler document for available runtime checks.

Defensive programming tips

- Consider the following example code problem.c

```c
int main()
{
   int a, b;
   int x1, x2;

   if (a = b)
     printf("%d\n", x1);
   return 0;
}
```

```
gcc –o problem problem.c
./problem
1074729080
```

Defensive programming tips

- Use –Wall function to help find errors at compile time

```
gcc –o -Wall problem problem.c
```

```
problem.c: In function main:
problem.c:6: warning: suggest parentheses around assignment used as truth value
problem.c:7: warning: implicit declaration of function printf
problem.c:7: warning: incompatible implicit declaration of built-in function printf
problem.c:4: warning: unused variable x2
```

Warning with line numbers are provided
Search in a good search engine to find details.

```
1        int main()
2        {
3          int a, b;
4          int x1, x2;
5
6        if (a = b)
7            printf("%d\n",
x1);
8          return 0;
9        }
```

Use assertion

- Provide one or more levels of instrumentation in your code for debugging. C programmers should take the advantage of the assert macro to ensure values fall within appropriate ranges

```
gcc –o –macro macro.c
./macro
macro: macro.c:12: main:
Assertion `n<=100' failed.
Abort

gcc –DNDEBUG –o macro macro.c
./macro
```

```c
#include <stdio.h>
#include <assert.h>

int main()
{
   int n;
   float x[100];
   n = 1000;

   /* Assert that n <= 100 */

   assert ( n <= 100);

   return 0;
}
```

Use assertion

- You can do similar things in Fortran
- It looks like a mixed code, how do we compile this?

```
ifort –DDEBUG –cpp macro.f

./a.out

Assertion (n<=100) is false File: macro.f Line 10
```

Fortran

```
 program main
       implicit none
       integer n
       real x(100)
       n = 1000

#ifdef DEBUG
       if ( n > 100) then
          print*,' Assertion (n <= 100) is false'
          print*,' File: ',__FILE__,' Line: ',__LINE__
          stop
       endif
#endif

       stop
       end
```

Defensive programming tips

- One of the best defenses against runtime bugs is to use basic defensive programming techniques:
  1. check all function return codes for errors
  2. check all input values controlling program execution to ensure they are within acceptable ranges
  3. echo all physical control parameters to a location that you will look at routinely (e.g. stdout). Better yet, save all parameters necessary to repeat an analysis in your solution files (netCFD and HDF5)
  4. in addition to monitoring for obvious floating point problems (e.g. division by zero), check for non-physical results in your simulations (e.g. supersonic velocities predicted by an incompressible flow solver)

Defensive programming tips

- Additional suggestions:
  1. maintain test cases for testing (code verification)

  2. use version control systems (e.g. git), which is to be discussed soon

  3. maintain a clean modular structure with documented interfaces

  4. add comments! your groupmates will thank you and it just may save your dissertation when you are revisiting a tricky piece of code after a year or two

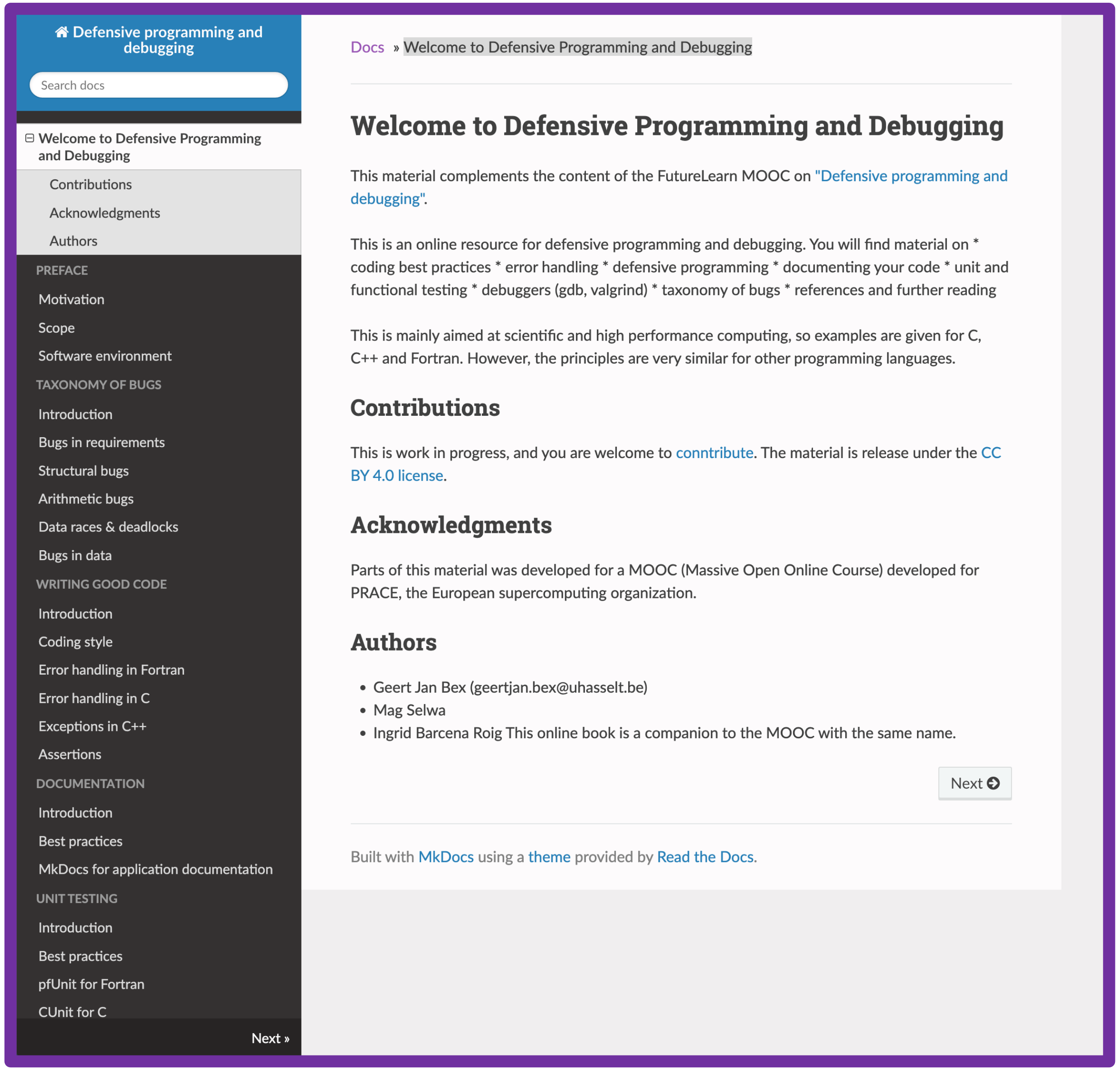
  5. strong error checking is the mark of a good programmer

References

- Reference:

“Welcome to Defensive Programming and Debugging”

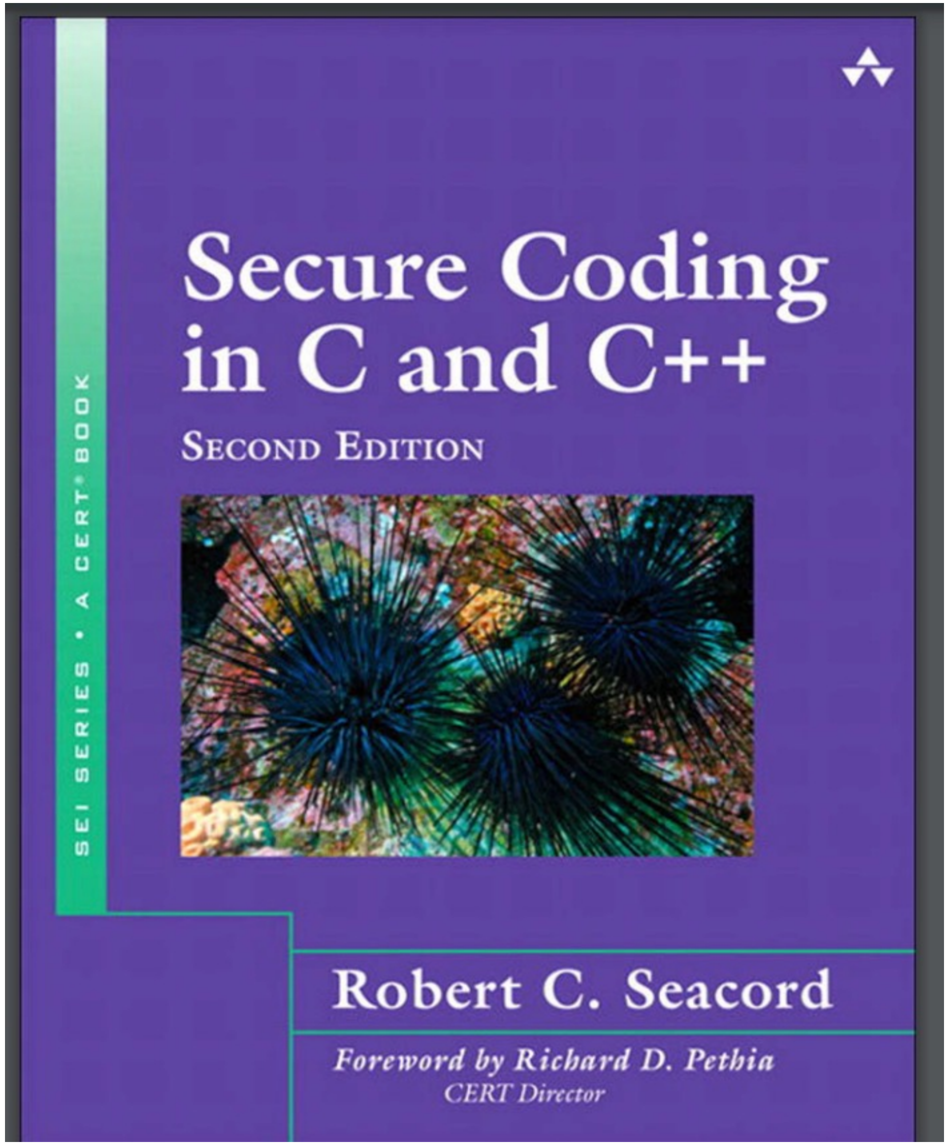
https://gjbex.github.io/DPD-online-book/



References

- Reference:

  “Secure Coding in C and C++”



GDB

Debugging process

- We recognize that defensive programming can greatly reduce debugging needs, but at some point, we all have to roll up our sleeves and track down a bug

- The basic steps in debugging are straightforward in principle:
  - ➢ recognize that a bug exists
  - ➢ isolate the source of the bug
  - ➢ identify the cause of the bug
  - ➢ determine a fix for the bug
  - ➢ apply the fix and test it

- In practice, these can be difficult for particularly pesky bugs; hence we need some more tools at our disposal (i.e. a debugger)

Standard debuggers

- Command line debuggers are powerful tools to aid in diagnosing problematic applications and are available on all UNIX architectures for C/C++ and Fortran

- Example debuggers: gdb, valgrind, lldb, etc.

- The basic use of these debuggers is as a front-end for stepping through your application and examining variables, arrays, function returns, etc. at different times during the execution

- Gives you an opportunity to investigate the dynamic runtime behavior of the application

Debugging basics

For effective debugging, a few commands need to be mastered:

- show program backtraces (the calling history up to the current point)
- set breakpoints
- display the value of individual variables
- set new values
- step through a program

GDB

GDB is the GNU project DeBugger

www.gnu.org/software/gdb

- A command line tool for debugging your code.
- Developed in 1986 by R. Stallman as part of his GNU project.
- Offers extensive facilities for tracing and altering the execution of computer programs.
- It is a command line tool without GUI support. There are front-ends built for it, such as DDD (data display debugger).

Debugging basics

- For debugging sessions, you should compile your application with extra debugging information included (e.g. the symbol table)

- The symbol table maps the binary execution calls back to the original source code definitions

- To include this information, add “-g” to your compilation directives:

```
gcc –g –o hello hello.c
```

- gdb can be started directly from the shell. You may include the name of the program to be debugged.

Running gdb

- gdb can be started directly from the shell.

- You may include the name of the program to be debugged, or with an optional core file

```
gdb
gdb a.out
gdb a.out corefile
```

spawns a new instance of ./a.out

examines trapped state in corefile

- gdb can also attach to a program that is already running; you just need to know the PID associated with the desired process

```
– gdb a.out 1134
```

useful if an application seems to be slow or stuck and you want to see what it is doing currently

gdb basics

Common commands for gdb:

- run – starts the program; if you do not set up any breakpoints, the program will run until it terminates or core dumps; program command line arguments can be specified here
- print – prints a variable located in the current scope
- next – executes the current command and moves to the next command in the program
- step – steps through the next command. Note: if you are at a function call, and you issue next, the the function will execute and return. However, if you issue step, then you will go to the first line of that function.
- break – set a breakpoint
- continue – used to continue till next breakpoint or termination

Note: shorthand notations exist for most of these commands: eg. 'c' = continue

gdb basics

More commands for gdb

- list – show code listing near the current execution location
- delete – delete a breakpoint
- condition – make a breakpoint conditional
- display – continuously display value
- undisplay – remove displayed value
- where – show current fuction stack trace
- help – display help text
- quit – exit gdb

GDB example 1

- Let us consider the following C code for subsequent examples (basic1.c)

```c
#include <stdio.h>

int main()
{
  int j = 3;
  int k = 7;
  j += k;
  k = j * 2;
  printf("Hello there \n");
  return 0;
}
```

gdb session

```
(gdb) run
Starting program: /work/mae-liuj/mae-5032/09-gdb-01/a.out
Hello there
[Inferior 1 (process 286278) exited normally]
Missing separate debuginfos, use: debuginfo-install glibc-2.17-222.el7.x86_64
(gdb) break main
Breakpoint 1 at 0x400525: file basic1.c, line 5.
(gdb) run
Starting program: /work/mae-liuj/mae-5032/09-gdb-01/a.out

Breakpoint 1, main () at basic1.c:5
5           int j = 3;
(gdb) next
6           int k = 7;
(gdb) list
1       #include <stdio.h>
2
3       int main()
4       {
5           int j = 3;
6           int k = 7;
7           j += k;
8           k = j * 2;
9           printf("Hello there \n");
10          return 0;
(gdb)
```

We use run to start the program in gdb. We may add argements after run just like how we run the code in shell (run arg1 arg2).

We may use break to set the breakpoint.

We have the next command to run the program and stop at the next line.

The list command will print the code where the gdb current at.

GDB session (cont.)

```
(gdb) where
#0  main () at basic1.c:6
(gdb) print j
$1 = 3
(gdb) next
7           j += k;
(gdb) print &j
$2 = (int *) 0x7fffffffce8c
(gdb) p j+k
$3 = 10
(gdb) p (j+3) + k - 2
$4 = 11
(gdb) p *(&k)
$5 = 7
(gdb) 
```

The where command will locate the position in the source code.

The print command is very useful. It shows the variable value. It also understand the expression in C syntax.

We may leave gdb by quit.

We may use r for run, n for next, p for print.

GDB example 2

- Let us consider the following C code for subsequent examples (basic2.c)

```c
#include "black_box.h"

void crash(int *i)
{
  *i = 1;
}

void f(int * i)
{
  int * j = i;
  j = complicated(j);
  j = sophisticated(j);
  crash(j);
}

int main()
{
  int i;
  f(&i);
  return 0;
}
```

```c
#include <stdlib.h>
int * complicated(int * j)
{
  return j;
}

int * sophisticated(int * j)
{
  return NULL;
}
```

```
mae-liuj::login01 { ~/mae-5032/09-gdb-02 }
-> gcc -g basic2.c
mae-liuj::login01 { ~/mae-5032/09-gdb-02 }
-> gdb a.out
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-110.el7
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /work/mae-liuj/mae-5032/09-gdb-02/a.out...done.
(gdb) run
Starting program: /work/mae-liuj/mae-5032/09-gdb-02/a.out

Program received signal SIGSEGV, Segmentation fault.
0x00000000004004f6 in crash (i=0x0) at basic2.c:5
5           *i = 1;
Missing separate debuginfos, use: debuginfo-install glibc-2.17-222.el7.x86_64
(gdb) break crash
Breakpoint 1 at 0x4004f2: file basic2.c, line 5.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /work/mae-liuj/mae-5032/09-gdb-02/a.out

Breakpoint 1, crash (i=0x0) at basic2.c:5
5           *i = 1;
```

compile with -g

set break at crash function

we see I is set to be a null pointer

GDB session (cont.)

```
(gdb) up
#1  0x000000000040053e in f (i=0x7fffffffce8c) at basic2.c:13
13          crash(j);
(gdb) up
#2  0x0000000000400554 in main () at basic2.c:19
19          f(&i);
(gdb) down
#1  0x000000000040053e in f (i=0x7fffffffce8c) at basic2.c:13
13          crash(j);
(gdb) down
#0  crash (i=0x0) at basic2.c:5
5           *i = 1;
```

we may use up and down to go to different levels of function calls.

We can also see that I enters f as an argument, and it is not a null pointer.

GDB session (cont.)

```
(gdb) break f
Breakpoint 1 at 0x40050a: file basic2.c, line 10.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /work/mae-liuj/mae-5032/09-gdb-02/a.out

Breakpoint 1, f (i=0x7fffffffce8c) at basic2.c:10
10          int * j = i;
(gdb) p j
$1 = (int *) 0x0
(gdb) n
11          j = complicated(j);
(gdb) p j
$2 = (int *) 0x7fffffffce8c
(gdb) n
12          j = sophisticated(j);
(gdb) p j
$3 = (int *) 0x0
(gdb) n
13          crash(j);
(gdb) p j
$4 = (int *) 0x0
```

We know something happened inside function f. Let us now set a breakpoint at f and run gdb again.

We may use print (p) and next to examine the values of j at each line of the code.

GDB session (cont.)

```
(gdb) display j
1: j = (int *) 0x0
(gdb) n
11          j = complicated(j);
1: j = (int *) 0x7ffffffce8c
(gdb) n
12          j = sophisticated(j);
1: j = (int *) 0x0
(gdb) n
13          crash(j);
1: j = (int *) 0x0
(gdb)
```

Alternatively, you may also use the display command with the next command. We see that everytime we run next, it prints the source code and the value of j.

After using display, we may use undisplay [id] to finish displaying variables.

GDB session (cont.)

```
(gdb) run
Starting program: /work/mae-liuj/mae-5032/09-gdb-02/a.out

Program received signal SIGSEGV, Segmentation fault.
0x00000000004004f6 in crash (i=0x0) at basic2.c:5
warning: Source file is more recent than executable.
5         *i = 1;
Missing separate debuginfos, use: debuginfo-install glibc-2.17-222.el7.x86_64
(gdb) backtrace
#0  0x00000000004004f6 in crash (i=0x0) at basic2.c:5
#1  0x000000000040053e in f (i=0x7fffffffce8c) at basic2.c:13
#2  0x0000000000400554 in main () at basic2.c:19
```

The command backtrace or bt is very useful as well. It will print a summary of how your program got where it is.

GDB example 3

- Let us consider the following C++ code for subsequent examples (basic3.c)

```
#include <iostream>

int factorial(const int &n)
{
  if(n != 0) return n * factorial(n-1);
  else return 1;
}

int main()
{
  int n;
  std::cout<<"Please enter a positive integer: \n";
  if(std::cin>>n && n>=0)
    std::cout<<n<<"! = "<<factorial(n)<<std::endl;
  else
    std::cout<<"That is not a positive integer!\n";
}
```

GDB session

```
(gdb) run
Starting program: /work/mae-liuj/mae-5032/09-gdb-03/a.out
Please enter a positive integer:
4

Breakpoint 1, factorial (n=@0x7fffffffce7c: 4) at basic3.c:5
warning: Source file is more recent than executable.
5           if(n != 0) return n * factorial(n-1);
Missing separate debuginfos, use: debuginfo-install glibc-2.17-222.el7.x86_64
6_64 libstdc++-4.8.5-28.el7.x86_64
(gdb) step

Breakpoint 1, factorial (n=@0x7fffffffce4c: 3) at basic3.c:5
5           if(n != 0) return n * factorial(n-1);
(gdb) continue
Continuing.

Breakpoint 1, factorial (n=@0x7fffffffce0c: 2) at basic3.c:5
5           if(n != 0) return n * factorial(n-1);
(gdb)
```

We use step or s to go into the function and start executing its code one line at a time. If the line of command does not involve function calls, step is equal to next.

We may use continue to run the program untill it hit the next breakpoint or finishes.

We have the finish command to finish the current function call and stop.

GDB example 4

- Let us consider the following C++ code for subsequent examples (basic4.c)

```cpp
#include <iostream>

int unknown(int &s)
{
  s += 1;
  return s;
}

void small(int &a)
{
  a /= 68;
}

int bar(int &p)
{
  p *= 3;
  return unknown(p);
}

void oof(int &n)
{
  n *= n - 20;
  n = n - bar(n);
}

void foo(int &z)
{
  z = bar(z);
  oof(z);
  small(z);
  ++z;
}

int main()
{
  int x = 10;
  foo(x);

  if(x != 0)
  {
    std::cerr<<" Error: x = "<<x<<", which is not 0.\n";
    return 3;
  }
}

```

I care how does the variable x varies in the functions.

GDB session

```
(gdb) break main
Breakpoint 1 at 0x400839: file basic4.c, line 36.
(gdb) run
Starting program: /work/mae-liuj/mae-5032/09-gdb-04/a.out

Breakpoint 1, main () at basic4.c:36
36          int x = 10;
Missing separate debuginfos, use: debuginfo-install glibc-2.17-
6_64 libstdc++-4.8.5-28.el7.x86_64
(gdb) p x
$1 = 0
(gdb) n
37          foo(x);
(gdb) p x
$2 = 10
(gdb)
```

We use break and print or s to locate that the issue is inside the function foo.

Of course we may jump into foo and add more breakpoints to observe the variables.

GDB session

```
(gdb) watch x
Hardware watchpoint 2: x
(gdb) continue
Continuing.
Hardware watchpoint 2: x

Old value = 10
New value = 30
bar (p=@0x7ffffffce7c: 30) at basic4.c:17
17          return unknown(p);
(gdb) list
12        }
13
14        int bar(int &p)
15        {
16          p *= 3;
17          return unknown(p);
18        }
19
20        void oof(int &n)
21        {
(gdb)
```

We use watch and continue to monitor the variation of the varialbe x.

GDB session

```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000400839 in main() at basic4.c:36
        breakpoint already hit 1 time
2       hw watchpoint  keep y                      x
        breakpoint already hit 3 times
(gdb) delete 2
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000400839 in main() at basic4.c:36
        breakpoint already hit 1 time
(gdb) c
Continuing.
 Error: x = -9, which is not 0.
```

We use info breakpoints to list the current breakpoints we set.

We can delete the breakpoints by the command delete.

GDB example 5

- Let us consider the following C++ code for subsequent examples (basic5.c)

```
#include "myfun.h"

int main()
{
   int x = 16;

   x = mystery(x);

   if(x%2 ==0) print_even();
   else print_odd();

   return 0;
}
```

```
#include <iostream>

void print_even()
{
  std::cout<<"I love GDB.\n";
}

void print_odd()
{
  std::cout<<"My code works great.\n";
}

int mystery(int &n)
{
  return n+1;
}
```

GDB session

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /work/mae-liuj/mae-5032/09-gdb-05/a.out

Breakpoint 1, main () at basic5.c:5
5           int x = 16;
(gdb) target record-full
(gdb) n
7           if(x%2 ==0) print_even();
(gdb) n
I love GDB.
10          return 0;
(gdb) rn
7           if(x%2 ==0) print_even();
(gdb) rn

No more reverse-execution history.
main () at basic5.c:5
5           int x = 16;
```

We use target record-full to enable gdb to track both forwardly and backwardly.

Then we may do both next and reverse-next (or rn) to move around.

There are reverse-step and reverse-continue commands.

GDB session

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /work/mae-liuj/mae-5032/09-gdb-05/a.out

Breakpoint 1, main () at basic5.c:5
5           int x = 16;
(gdb) target record-full
(gdb) n
7           if(x%2 ==0) print_even();
(gdb) n
I love GDB.
10          return 0;
(gdb) rn
7           if(x%2 ==0) print_even();
(gdb) rn

No more reverse-execution history.
main () at basic5.c:5
5           int x = 16;
```

We use target record-full to enable gdb to track both forwardly and backwardly.

Then we may do both next and reverse-next (or rn) to move around.

There are reverse-step and reverse-continue commands.

GDB session

```
Breakpoint 1, main () at basic5.c:5
5           int x = 16;
(gdb) n
7           x = mystery(x);
(gdb) p x
$1 = 16
(gdb) set var x=15
(gdb) p x
$2 = 15
(gdb) n
9           if(x%2 ==0) print_even();
(gdb) n
I love GDB.
12          return 0;
(gdb)
```

We can use set var to change the variable value within gdb.

Valgrind

Valgrind

Valgrind is a programming tool for memory debugging, memory leak detection, and profiling.

There are multiple tools inside Valgrind:

- Memcheck detects memory management problems, where all reads and writes of memory are checked.
- Cachegrind is a cache profiler that performs detailed simulation of the L1 and L2 caches.
- Callgrind is an extension of cacegrind to procude extra information about callgraphs.

Valgrind example

- Let us consider the following C code for subsequent examples (main.c)

```c
#include <stdlib.h>

void f(void)
{
   int * x = malloc(10 * sizeof(int));
   x[10] = 0;
}

int main(void)
{
   f();
   return 0;
}
```

Valgrind session

- Compile the code with –g is better. You may compile with –O1 but the error message may be inaccurate.
- Put valgrind before running the executable.

```
gcc -g main.c
mae-liuj::login01 { ~/mae-5032/10-valgrind }
-> valgrind ./a.out
==99917== Memcheck, a memory error detector
==99917== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==99917== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==99917== Command: ./a.out
==99917==
==99917== Invalid write of size 4
==99917==    at 0x40053B: f (main.c:6)
==99917==    by 0x40054B: main (main.c:11)
==99917==  Address 0x5203068 is 0 bytes after a block of size 40 alloc'd
==99917==    at 0x4C29BC3: malloc (vg_replace_malloc.c:299)
==99917==    by 0x40052E: f (main.c:5)
==99917==    by 0x40054B: main (main.c:11)
==99917==
```

Valgrind session

```
gcc -g main.c
mae-liuj::login01 { ~/mae-5032/10-valgrind }
-> valgrind ./a.out
==99917== Memcheck, a memory error detector
==99917== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==99917== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==99917== Command: ./a.out
==99917==
==99917== Invalid write of size 4
==99917==    at 0x40053B: f (main.c:6)
==99917==    by 0x40054B: main (main.c:11)
==99917==  Address 0x5203068 is 0 bytes after a block of size 40 alloc'd
==99917==    at 0x4C29BC3: malloc (vg_replace_malloc.c:299)
==99917==    by 0x40052E: f (main.c:5)
==99917==    by 0x40054B: main (main.c:11)
==99917==
```

Valgrind session (cont.)

definitely lost: your progoram is leaking memory for sure. You have to fix it!

```
==98251== LEAK SUMMARY:
==98251==    definitely lost: 40 bytes in 1 blocks
==98251==    indirectly lost: 0 bytes in 0 blocks
==98251==      possibly lost: 0 bytes in 0 blocks
==98251==    still reachable: 0 bytes in 0 blocks
==98251==         suppressed: 0 bytes in 0 blocks
==98251== Rerun with --leak-check=full to see details of leaked memory
==98251==
==98251== For counts of detected and suppressed errors, rerun with: -v
==98251== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

possibly lost: your progoram is leaking memory unless there are some funny things with pointers.

Summary

- People are bad at writing code. We need to develop a good coding style or defensive programming for our development job.

- GDB and Valgrind are quite powerful tools, and we have only scratched the surface today.

- “Debugging with GDB” by R. Stallman, et al.

- “The art of debugging with GDB, DDD, and Eclipse” by N. Matloff and P.J. Salzman; 中译本 《软件调试的艺术》

- Valgrind has a great on-line user manual: https://valgrind.org/docs/manual/manual.html