

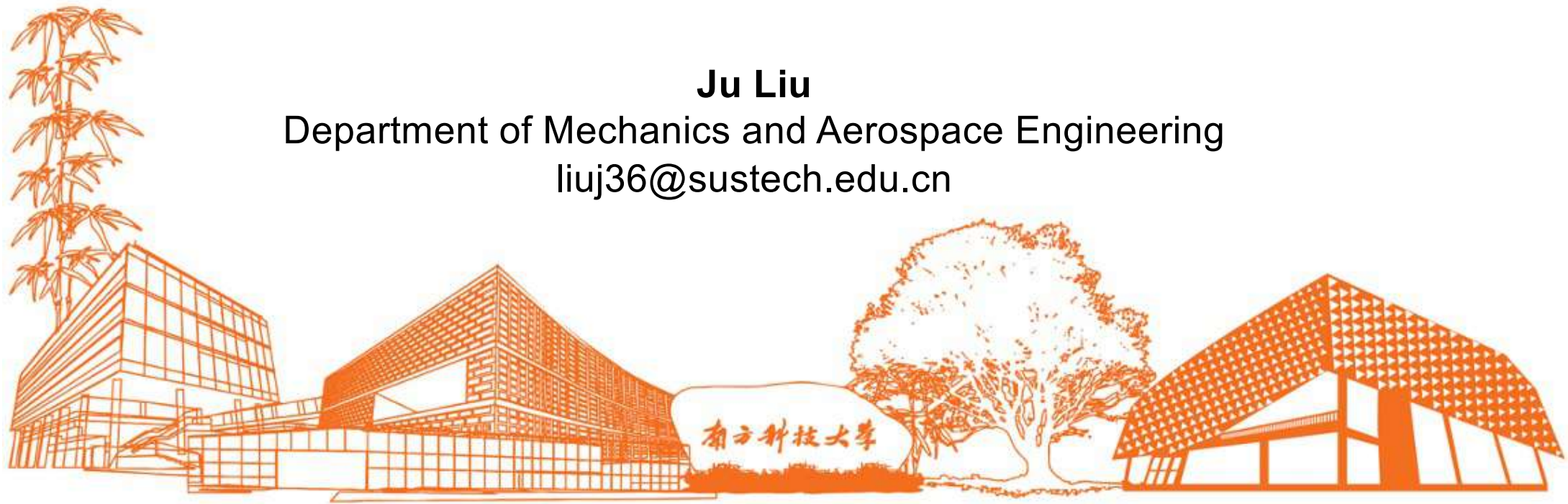
MAE 5032 High Performance Computing: Methods and Practices

Lecture 8: Defensive programming and debugging

Ju Liu

Department of Mechanics and Aerospace Engineering

liuj36@sustech.edu.cn



Defensive Programming

“The best code explains itself”

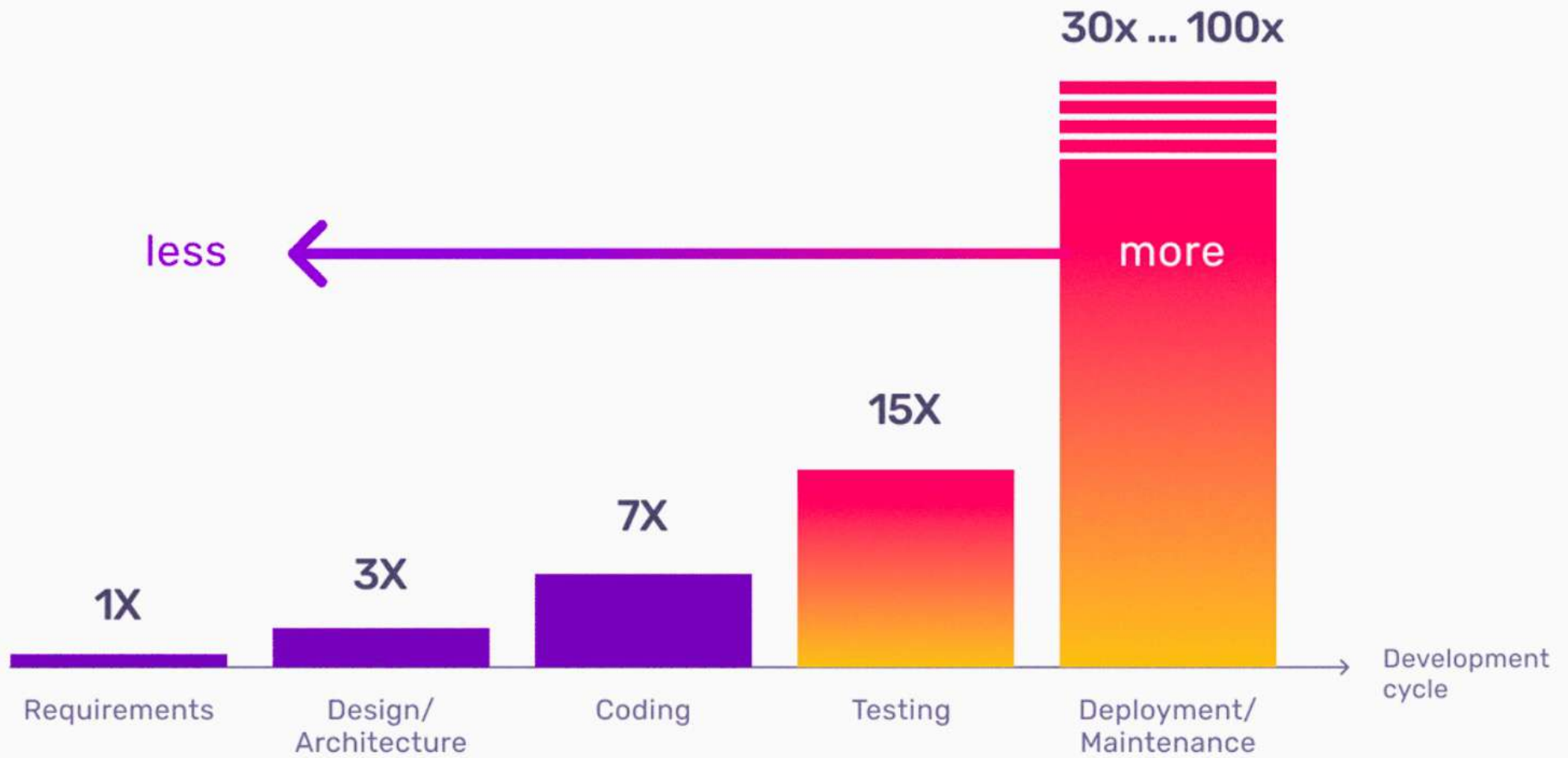


**“80% of the lifetime cost of a piece of software
goes to maintance”**



**UNREAL
ENGINE**

Cost of software defects

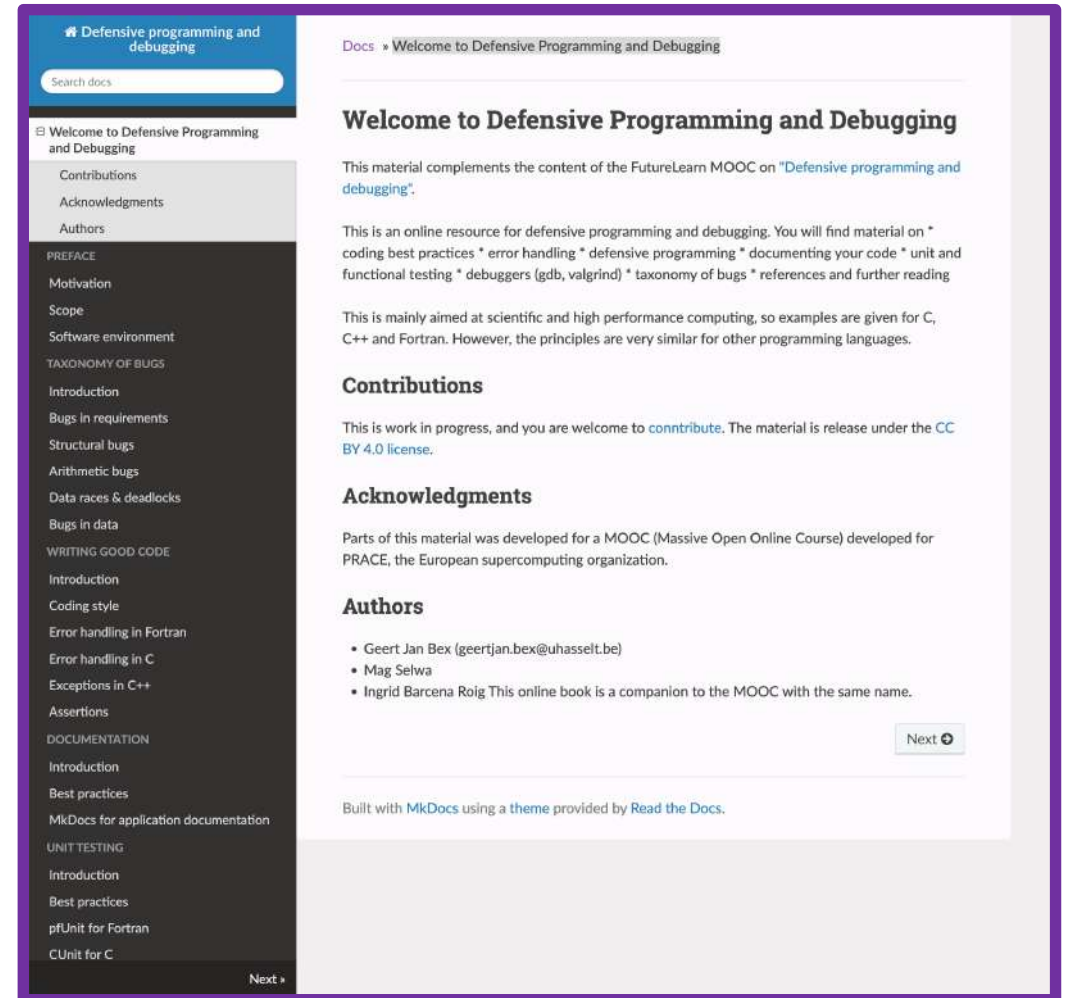


References

- Reference:

“Welcome to Defensive Programming and Debugging”

https://gjbex.github.io/Defensive_programming_and_debugging/



Codes and bugs



G. Hopper seated at the input console for the UNIVAC I.

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
 - Commercial applications have bugs too

An example of literal debugging from Harvard Mark II as recorded by G. Hopper: a moth found in the machine has been taped to the logbook.



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
 - Commercial applications have bugs too
- Bugs have huge impact on our society
 - In 1962, the Mariner-1 space probe mission failed due to a missing overbar in the source code, causing a loss of 18.5 million dollars.



https://en.wikipedia.org/wiki/Mariner_1

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
 - Commercial applications have bugs too
- Bugs have huge impact on our society
 - In 2009, a bug in the anti-lock-brake 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.

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: loop termination
 - logic
 - processing
 - initialization

```
> seq01
AACGTACCT
CCAGGT
> seq02
GGACAGGTT
AGGCTAAGC
TC
> seq03
CGGAC
```

```
make
run:

seq01: A: 4, C = 5, G = 3, T = 3
seq02: A: 5, C = 4, G = 7, T = 4
```

There is no tools that can help you detect this type of bug;
A comprehensive unit and functional tests is needed!

Structural bugs

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

This function will return 1 for negative input.

A code path to cover this case is not implemented.

A comprehensive test may be able to pick up on such issues.

```
integer function factorial(n)
  implicit none
  integer, intent(in) :: n
  integer :: i
  factorial = 1
  do i = 2, n
    factorial = factorial(i)
  end do
end function factorial
```

Structural bugs

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

Note that the description of what the subroutine is supposed to do (at the boundary value) is ambiguous.

We need proper testing to detect this type of bug.

```
if (x < low) then
    print '(A)', 'low'
else if (low < x .and. x < high)
    then
        print '(A)', 'medium'
else
    print '(A)', 'high'
end if
```


Structural bugs

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

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

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

Many bugs are caused when **negation** is involved.

Structural bugs

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

```
if (!(data = (double *) malloc(n*sizeof(double))))  
{  
    // handle memory issue  
}
```

```
double *data = (double *) malloc(n*sizeof(double));  
if (data == NULL)  
{  
    // handle memory issue  
}
```

Many bugs are caused when **negation** is involved.

Structural bugs

- This is a very broad category that can be divided into a number of subcategories
 - control flow
 - logic
 - processing
 - memory leaks (can be detected by valgrind and other inspectors)
 - 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

- control flow
- logic
- processing
- initialization

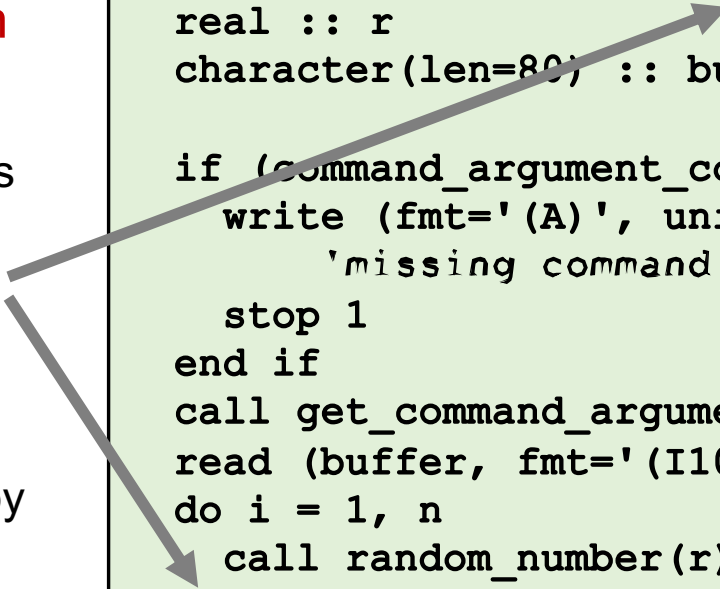
The variable data has been declared to be **allocatable**, but the **allocate statement is missing**.

Often can be found by compiler or valgrind.

explicit initializing data is good practice.

```
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
```



Arithmetic bugs

- Integers typically are held by 32 bits (4 bytes).
- 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: there is a set of waiting processes, $P = \{P1, P2, \dots, PN\}$, 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 users.
- Validate the input data
- Don't reinvent the wheels when a library is available for loading data (XML, YAML data format)
- Be careful with data conversion
narrowing conversion (-Wconversion)

```
#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

Defensive programming

- **Coding styles** are common guidelines to improve the readability, maintainability, prevent common errors, and make the code more uniform
- A **consistent code** base helps developers better understand code organization, focus on program logic, and reduce the time spent on interpreting other engineers' intentions.
- Popular coding styles
 - LLVM coding standards (<https://llvm.org/docs/CodingStandards.html>)
 - Google C++ style guide (<https://google.github.io/styleguide/cppguide.html>)
- Static analysis tools
 - clang-tidy (<https://clang.llvm.org/extra/clang-tidy/>)
 - cpp-checks (<https://sourceforge.net/p/cppcheck/wiki/ListOfChecks/>)

Defensive programming

- “Code formatting is about communication, and communication is the professional developer’s first order of business.”
 - separate commands, operators, by a space
 - prefer alignment
 - use the same indentation style
 - limit line length to be at most 80 characters long (help read the code in terminal)

```
cmdh5w->write_doubleScalar(  "fl_density",      fluid_density);
cmdh5w->write_doubleScalar(  "fl_mu",           fluid_mu);
cmdh5w->write_doubleScalar(  "wall_density",     wall_density);
cmdh5w->write_doubleScalar(  "wall_poisson",     wall_poisson);
cmdh5w->write_doubleScalar(  "wall_kappa",       wall_kappa);
cmdh5w->write_doubleScalar(  "init_step",        initial_step);
cmdh5w->write_intScalar(     "sol_record_freq",  sol_record_freq);
cmdh5w->write_intScalar(     "nqp_tet",         nqp_tet);
cmdh5w->write_intScalar(     "nqp_tri",         nqp_tri);
cmdh5w->write_string(        "lpn_file",        lpn_file);
cmdh5w->write_string(        "date",            SYS_T::get_date() );
cmdh5w->write_string(        "time",            SYS_T::get_time() );
cmdh5w->write_string(        "petsc-version",    PETSc_T::get_version() );

cmdh5w->write_intScalar(     "inflow_type",      inflow_type);
cmdh5w->write_string(        "inflow_file",      inflow_file);
if( inflow_type == 1 )
    cmdh5w->write_doubleScalar("inflow_thd_time", inflow_thd_time );
```

```
33
34 // Update is called once per frame
35 void Update()
36 {
37     Color player_icon_col = new Color(1,1,1,1);
38     Color box_col = new Color(1,1,1,1);
39     if (!manager.player_ready[id_num]){
40         float grey = 0.3f;
41         for (int x=0; x<5; x++){
42             for (int y=0; y<5; y++){
43                 player_icon_col = new Color(grey, grey, grey, 0.4f);
44                 box_col = new Color(1,1,1, 0.7f);
45             }
46         }
47     }
48     sprite.color = player_icon_col;
49     box.color = box_col;
50     skull_sprite.color = box_col;
51 }
52
53
```

Defensive programming

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



fac *= i

Defensive programming tips

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

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

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

Descriptive names

“Beyond basic mathematical aptitude, the difference between good programmers and great programmers is verbal ability”

Marissa Mayer

- Naming is hard. Most of the time, code is shared with other developers. It is worth spending a few seconds to find the right name.
- Adopt names commonly used in real contexts.

Descriptive names

- Variable names should be nouns;
 - avoid using single letter for variable
- Function names should be verbs;
- If a function return a property, it should be phrased as a question
 - `is_xxx()`, `has_xxx()`, etc.
- Limit the scope of variables.
 - In old days, you have to declare all variables at the start of a block.
 - 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.
- Use `true/false` for boolean variables instead of numeric values `0/1`

Descriptive names

- Use natural word pair,
 - create()/destroy(),
 - open()/close(),
 - begin()/end(),
 - source()/destination(),
 - set()/get()
- Abbreviations are generally bad, longer names are better in most cases.

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

- Comment
 - Comments shall not be a substitute for code that is easy to understand
 - keep it up-to-date with the code
 - do not disable code fragments by comment, use git.

Error handling

- Checking errors is not a waste of time, and it saves 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);  
}
```

C

```
real, allocatable :: p(:)  
integer :: nnodes  
  
allocate(p(0:nnodes), stat=ierr)  
if (ierr /= 0) then  
    print *, "Allocation error for p!"  
    stop 1  
end if
```

Fortran

Hardening Techniques

- **Hardening techniques** are compiler and linker options that enhance the security and reliability of applications by mitigating vulnerabilities such as memory safety issues, undefined behavior, etc.
- Compiler Options Hardening Guide for C and C++
(<https://best.openssf.org/Compiler-Hardening-Guides/Compiler-Options-Hardening-Guide-for-C-and-C++.html>)
- Hardened mode of standard library implementations
(<https://medium.com/@simontoth/daily-bit-e-of-c-hardened-mode-of-standard-library-implementations-18be2422c372>)

Hardening Techniques

- `-Wstack-usage=<byte-size>`
Warn if the stack usage of a function might exceed byte-size. The computation done to determine the stack usage is conservative (no VLA)
- `-fstack-usage`
Makes the compiler output stack usage information for the program, on a per-function basis
- `-Wvla`
Warn if a variable-length array is used in the code
- `-Wvla-larger-than=<byte-size>`
Warn for declarations of variable-length arrays whose size is either unbounded, or bounded by an argument that allows the array size to exceed byte-size bytes

Hardening Techniques

- `-Wall`
Enable many standard warnings (~50 warnings)
- `-Wextra`
Enable some extra warning flags that are not enabled by `-Wall` (~15 warnings)
- `-Wpedantic`
Issue all the warnings by strict ISO C/C++
- `-Werror`
Treat warnings as errors
- `-Weverything`
Enable all warnings, only clang

Defensive programming

- Consider the following example code problem.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.

```
int main()  
{  
    int a, b;  
    int x1, x2;  
  
    if (a = b)  
        printf("%d\n", x1);  
    return 0;  
}
```

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
```

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

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

    /* Assert that n <= 100 */
    assert ( n <= 100 );

    return 0;
}
```

C

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
- You can quickly locate the error with the assertion message.
- You can turn the assertion off by **-DNDEBUG** in gcc.
- Some code has their own assertion programmed.

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

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

    /* Assert that n <= 100 */
    assert ( n <= 100 );

    return 0;
}
```

C

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
- You can quickly locate the error with the assertion message.
- You can turn the assertion off by **-DNDEBUG** in gcc.
- Some code has their own assertion programmed.

```
#if PETSC_DEFINED(USE_DEBUG) C
    #define ASSERT(cond, message, ...)
        SYS_T::print_fatal_if_not(cond, message,
                                   ##__VA_ARGS__)
#else
    #define ASSERT(cond, ...) ((void)0)
#endif
```

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
```

```
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
```

Fortran

Unit test

- “A unit test is an automated piece of code that invokes a unit of work and checks a single assumption about its behavior.” – Kent Beck, The Art of Unit Testing
- You will typically write tests that compares the return value of a function to an expected value.

```
#include <cassert>

int add(int a, int b) { return a + b; }

int main() {
    assert(add(2, 3) == 5);
    assert(add(0, 0) == 0);
    assert(add(-1, 1) == 0);
    return 0;
}
```

Unit test

- “A unit test is an automated piece of code that invokes a unit of work and checks a single assumption about its behavior.” – Kent Beck, The Art of Unit Testing
- You will typically write tests that compares the return value of a function to an expected value.
- Not only check for **normal** cases, but also for **edge** or **corner** cases.
- Not only check for normal behaviors, but also for **exceptions/error handlings**.
- Check for **all branches** if there are conditional statement in the function.

Unit test


- Test-driven development (TDD): defining automated functional tests before implementing the functionality.
- The process consists of the following steps:
 1. Write a test for a new functionality
 2. Write the minimal code to pass the test
 3. Improve/refactor the code iterating with the test verification
 4. Go to step 1.
- Main advantages:
 - Small debugging cost
 - Understandable behavior: new user can learn how the system works from tests
 - Increase confidence: developers are more confident that their code will work as intended as it has been extensively tested
 - Faster development

Defensive programming

- One of the best defenses against runtime bugs is to use basic defensive programming techniques:
 - check (*all*) function return codes for errors

```
int METIS_PartMeshDual(idx_t *ne, idx_t *nn, idx_t *eptr, idx_t *eind,
    idx_t *vwgt, idx_t *vsize, idx_t *ncommon, idx_t *nparts,
    real_t *tpwgts, idx_t *options, idx_t *objval, idx_t *epart,
    idx_t *npart)
{
    int sigrval=0, renumber=0, ptype;
    idx_t i, j;
    idx_t *xadj=NULL, *adjncy=NULL, *nptr=NULL, *nind=NULL;
    idx_t ncon=1, pnunflag=0;
    int rstatus = METIS_OK;

    /* set up malloc cleaning code and signal catchers */
    if (!gk_malloc_init())
        return METIS_ERROR_MEMORY;
}
```



```
if(metis_result != METIS_OK)
{
    std::cerr<<"ERROR: PARTITION FAILED: "<<std::endl;
    switch( metis_result)
    {
        case METIS_ERROR_INPUT:
            std::cout << "METIS_ERROR_INPUT" << std::endl;
            break;
        case METIS_ERROR_MEMORY:
            std::cout << "METIS_ERROR_MEMORY" << std::endl;
            break;
        case METIS_ERROR:
            std::cout << "METIS_ERROR" << std::endl;
            break;
        default:
            break;
    }
    exit(1);
}
```

Defensive programming

- 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

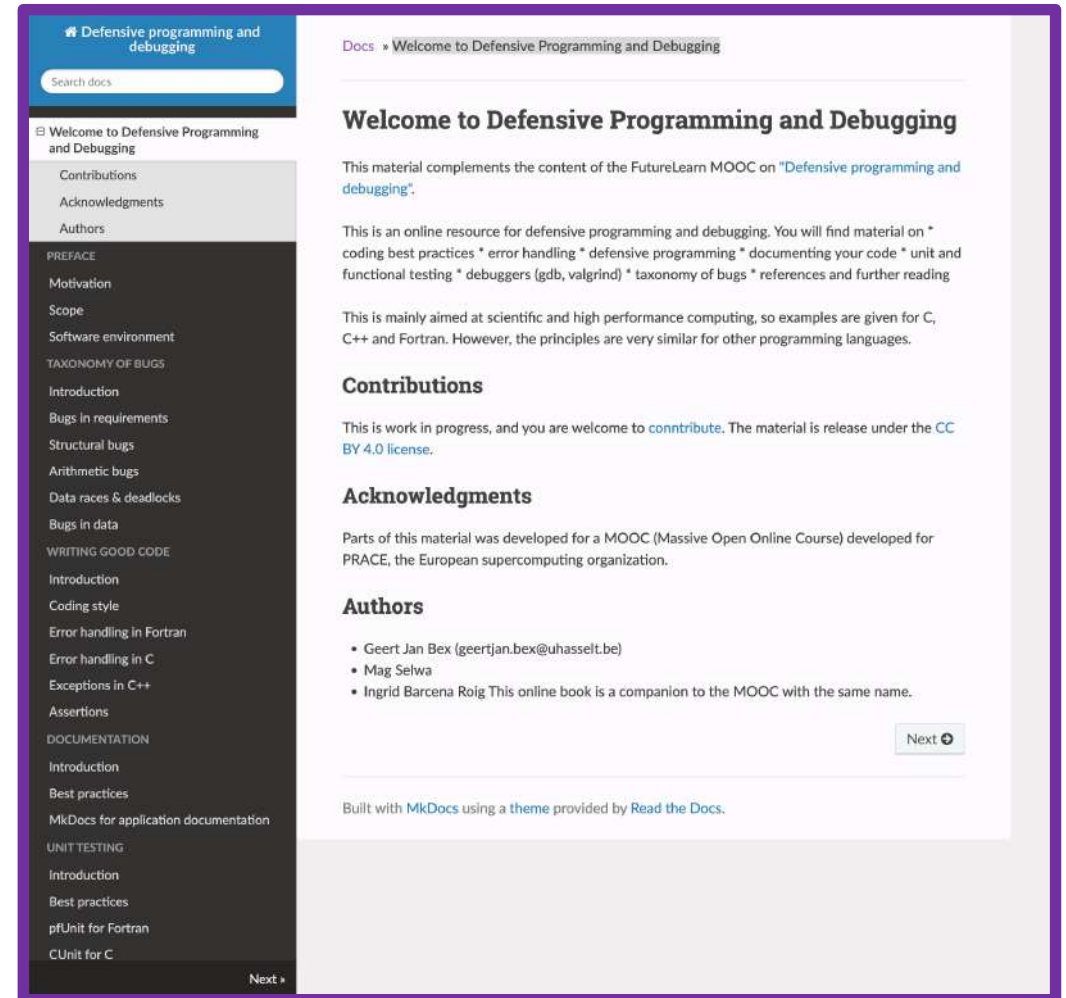
- Additional suggestions:
 1. maintain test cases for testing (code verification);
 2. use version control systems (e.g. git), which is to be discussed soon;
 3. maintain a clean modular structure with documented interfaces;
 4. add comments. Your groupmates will thank you and it just may save your dissertation 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:

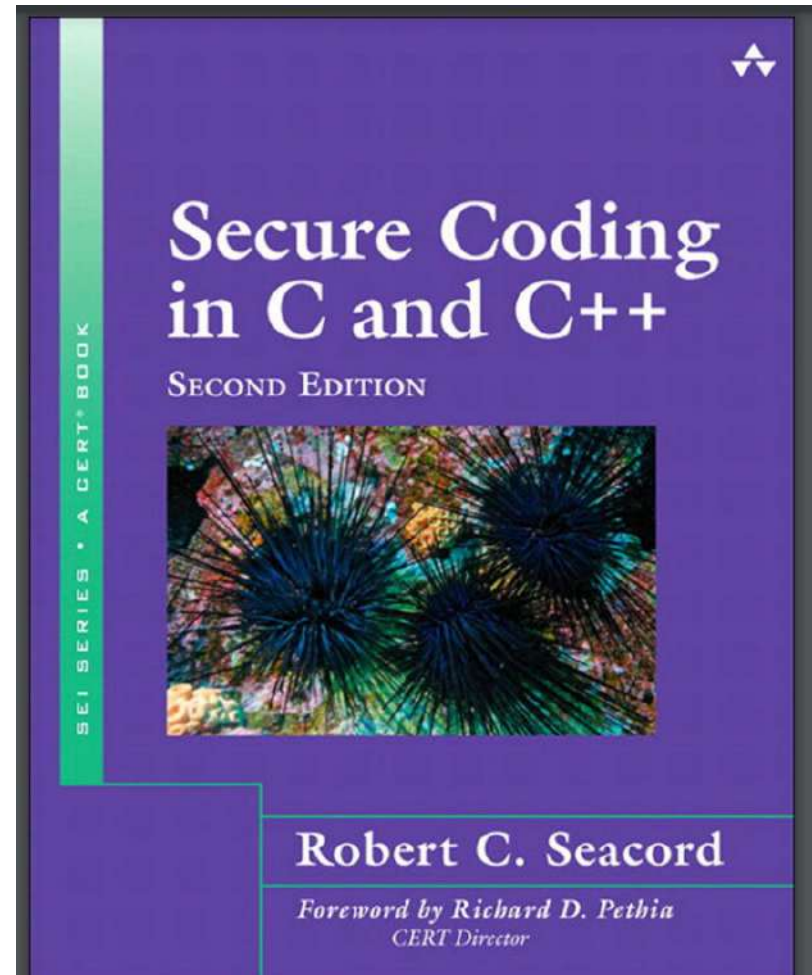
“Welcome to Defensive Programming and Debugging”

https://gjbex.github.io/Defensive_programming_and_debugging/



References

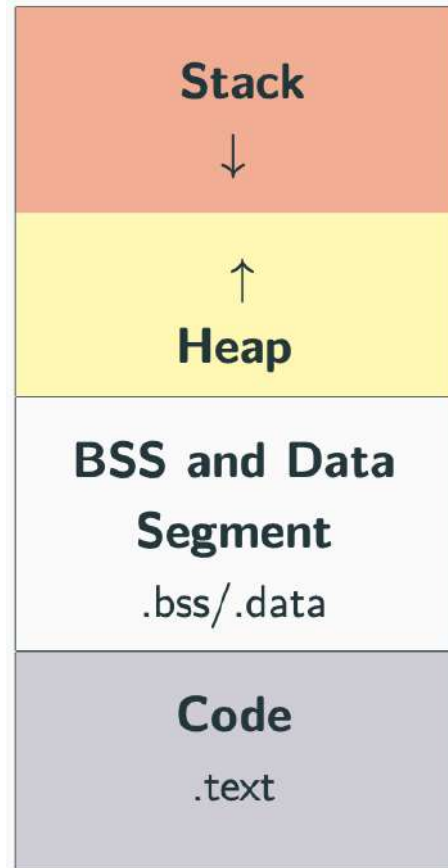
- Reference:
“Secure Coding in C and C++”



Heap & Stack

Process address space

higher memory address
0x00FFFFFF



Stack memory: `int data[10]`

dynamic memory: `new int[10];`
`malloc(10);`

static/global data: `int data [10];`

lower memory address
0x00FF0000

BSS and Data segments



BSS Segment (.bss): Uninitialized global and static variables

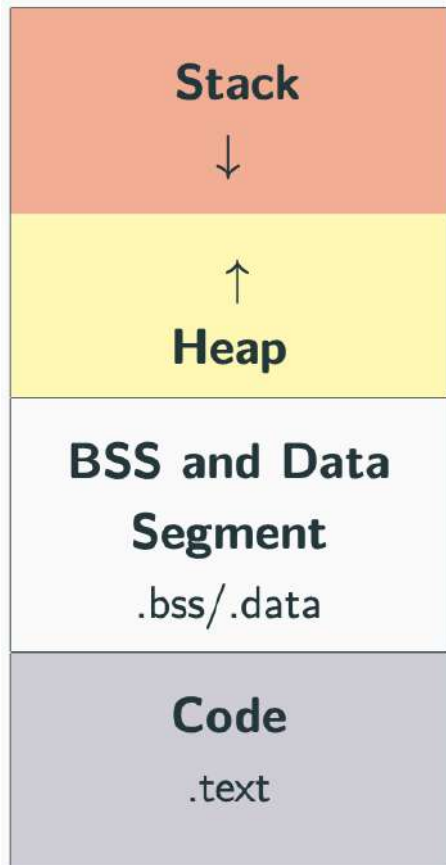
Data Segment (.data): Initialized global and static variables.

The BSS/Data segments are larger than stack (max ~ 1GB), but slower

```
int data [] = {1, 2};           // DATA segment memory
int big_data[1000000] = {};     // BSS segment memory
                                // (zero initialized)

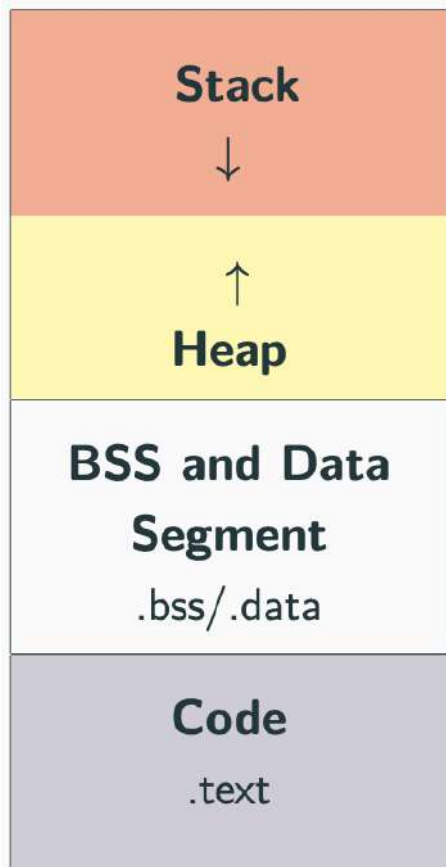
int main()
{
    int A[] = {1, 2, 3};        // stack memory
}
```

Stack and Heap memory



Aspect	Stack	Heap
Memory Organization	Contiguous (LIFO)	Contiguous within an allocation, fragmented between allocations (relies on virtual memory)
Max Size	Small (8MB on Linux, 1MB on Windows)	Whole system memory
If Exceed	Program crash at function entry (hard to debug)	Exception or nullptr
Allocation	Compile-time	Run-time
Locality	High	Low
Thread View	Each thread has its own stack	Shared among threads

Stack and Heap memory



A local variable is either in the stack memory or CPU register

The organization of the stack memory enables much higher performance.

This memory space is limited.

```
int data [] = {1, 2};           // DATA segment memory
int big_data[1000000] = {};     // BSS segment memory
                                // (zero initialized)

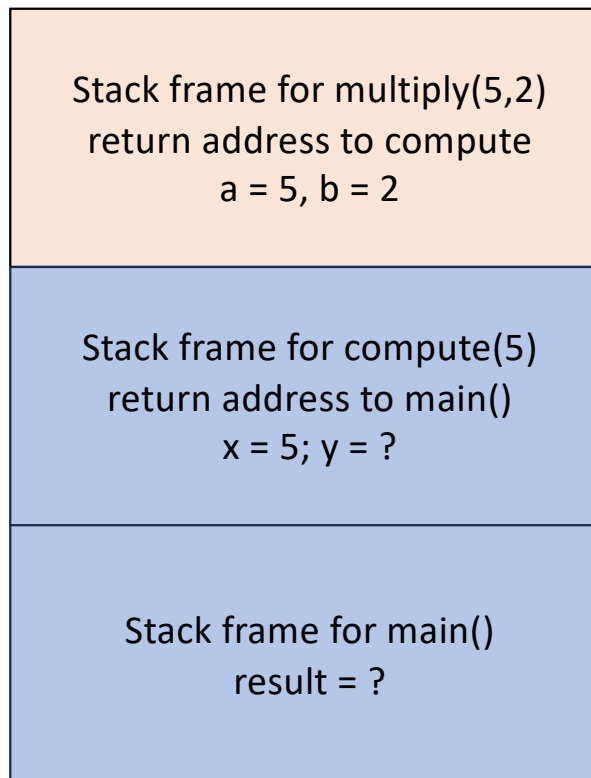
int main()
{
    int A[] = {1, 2, 3};        // stack memory
    char z[] = "abc";           // on stack
    void* ptr = malloc(4);      // variable "ptr" is on the stack
}
```


Call stack

```
int multiply(int a, int b)
{
    return a * b;
}

int compute(int x)
{
    int y = multiply(x, 2);
    return y + 1;
}

int main()
{
    int result = compute(5);
    return 0;
}
```

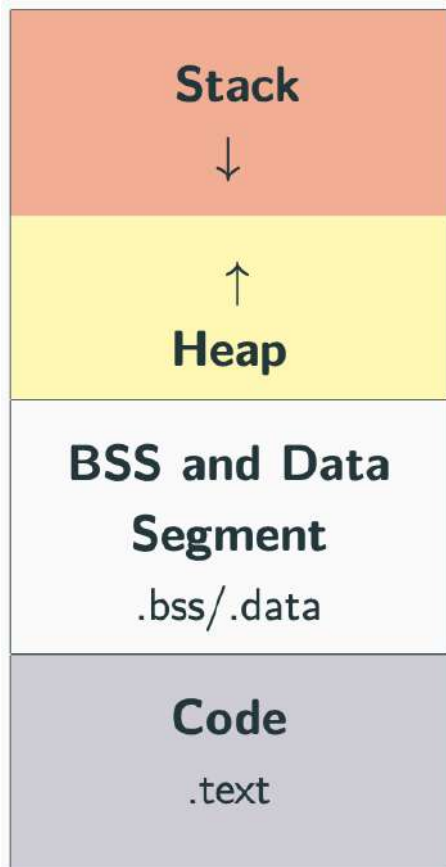


each box is known as a **stack frame**

a stack frame contains all the info needed to manage a function call, including: return address, input parameters, local variables, etc.

stack overflow: too much function calls

Stack and Heap memory



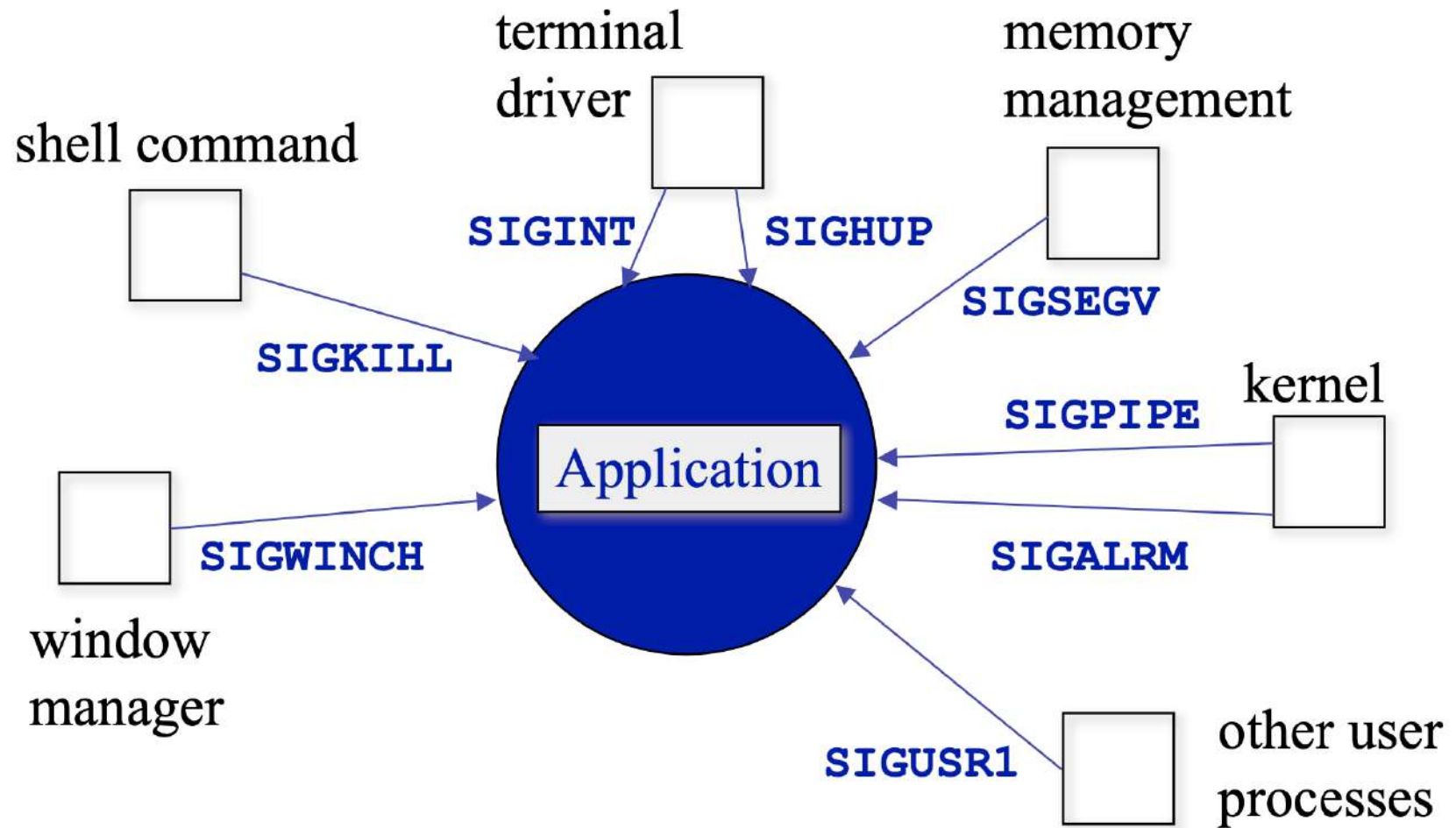
Heap memory are managed by the `new/delete` keywords.

A **memory leak** is a dynamically allocated entity in the heap memory that is no longer used by the program, but still maintained over its execution.

- illegal memory access -> segmentation fault/wrong results
- undefined behavior
- additional memory consumption

Singnal

Common signal sources



Common signal types

Name	Description	Default Action
SIGINT	Interrupt character typed	terminate process
SIGQUIT	Quit character typed (^\\)	create core image
SIGKILL	Kill signal	terminate process
SIGFPE	Floating exception	create core image
SIGSEGV	Invalid memory reference	create core image
SIGPIPE	Write on pipe but no reader	terminate process
SIGALRM	alarm() clock 'rings'	terminate process
SIGUSR1	user-defined signal type	terminate process
SIGUSR2	user-defined signal type	terminate process

Core dumps

- Recall that the default action for a SIGSEGV was to create a core image and abort.
- What is a core image?
 - a **core dump** is a record of the raw contents of one or more regions of working memory for an application at a given time
 - commonly used to debug a program that has terminated abnormally
 - main benefit of a core file is that a post-mortem analysis can be performed on an application that failed. If the symbol table was included, you can backtrace to exactly where the application when the exception occurred.
 - the location of an exception may not be the original location of a bug (particularly for memory bugs)

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

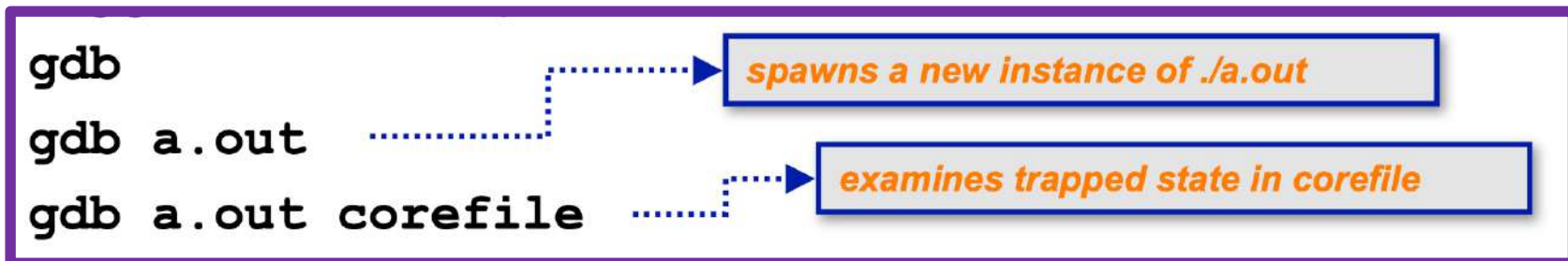
- For debugging sessions, you should compile your application with extra debugging information included (e.g. the symbol table)
- The symbol table maps the binary execution calls back to the original source code definitions
- To include this information, add “-g” to your compilation directives:

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



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 function stack trace
- **help** – display help text
- **quit** – exit gdb

GDB example 1

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

```
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;
11 }
```


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 arguments 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.

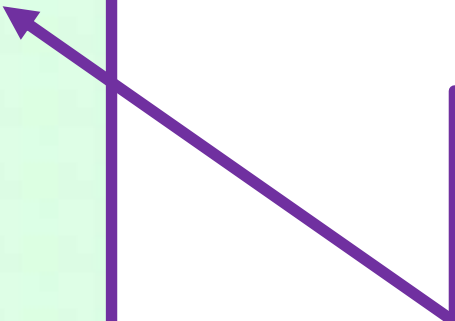
GDB example 2

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

```
1 #include "black_box.h"
2
3 void crash(int *i)
4 {
5     *i = 1;
6 }
7
8 void f(int * i)
9 {
10     int * j = i;
11     j = complicated(j);
12     j = sophisticated(j);
13     crash(j);
14 }
15
16 int main()
17 {
18     int i;
19     f(&i);
20     return 0;
21 }
```

```
#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
```

compile with -g

```
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
```

set break at crash function

```
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
```

we see i is set to be a null pointer

```
5      *i = 1;
```

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 *) 0x7fffffffce8c
(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)

```
1 #include <iostream>
2
3 int factorial(const int &n)
4 {
5     if(n != 0) return n * factorial(n-1);
6     else return 1;
7 }
8
9 int main()
10 {
11     int n;
12     std::cout<<"Please enter a positive integer: \n";
13     if(std::cin>>n && n>=0)
14         std::cout<<n<<"! = "<<factorial(n)<<std::endl;
15     else
16         std::cout<<"That is not a positive integer!\n";
17 }
```

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 until it hits 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)

```
1 #include <iostream>
2
3 int unknown(int &s)
4 {
5     s += 1;
6     return s;
7 }
8
9 void small(int &a)
10 {
11     a /= 68;
12 }
13
14 int bar(int &p)
15 {
16     p *= 3;
17     return unknown(p);
18 }
19
20 void oof(int &n)
21 {
22     n *= n - 20;
23     n = n - bar(n);
24 }
25
26 void foo(int &z)
27 {
28     z = bar(z);
29     oof(z);
30     small(z);
31     ++z;
32 }
33
34 int main()
35 {
36     int x = 10;
37     foo(x);
38
```

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

```
39     if(x != 0)
40     {
41         std::cerr<<" Error: x = "<<x<<" , which is not 0.\n";
42         return 3;
43     }
44 }
45
```

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=@0x7fffffffce7c: 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 variable 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)

```
1 #include "myfun.h"
2
3 int main()
4 {
5     int x = 16;
6
7     x = mystery(x);
8
9     if(x%2 ==0) print_even();
10    else print_odd();
11
12    return 0;
13 }
```

```
#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

Memory vulnerabilities

“70% of all vulnerabilities in Microsoft products are memory safety issues”

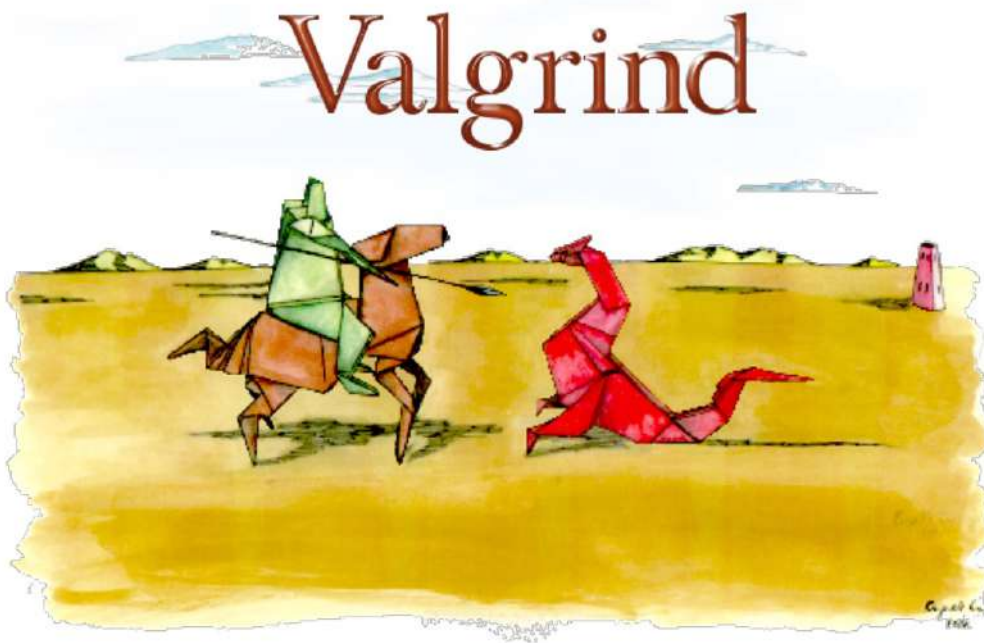
“Memory unsafety in Apple’s OS represents 66.3%-88.2% of all the vulnerabilities”

“Out-of-bounds reads/writes comprise 70% of all the vulnerabilities in Android”

Terms like buffer overflow, race condition, page fault, null pointer, stack exhaustion, heap exhaustion/corruption, use-after-free, double-free all describe **memory safety vulnerabilities**.

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 cachegrind to produce extra information about callgraphs.
- etc.

Valgrind

Install the latest version:

```
wget ftp://sourceware.org/pub/valgrind/valgrind-3.21.tar.bz2
tar xf valgrind-3.21.tar.bz2
cd valgrind-3.21
./configure --enable-lto
make -j 12
sudo make install
```

In linux, apt install valgrind works, but it could be an old version.

In Taiyi, you can use module load valgrind.

Valgrind

- When a program dynamically allocates memory and forgets to later free it, it creates a **leak**.
 - it typically won't cause a program to misbehave or give wrong answers
 - it may lead to run out-of-memory
- Memory error includes reading uninitialized memory, writing past the end of a piece of memory, accessing freed memory, etc.
 - it is a red flag
 - it is urgent and should be resolved immediately
- Valgrind memcheck is rather effective in detecting both issues

Valgrind example

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

```
#include<stdlib.h>
#include<stdio.h>
#include<time.h>

const int ARR_SIZE = 1000;

int main() {
    int *intArray = malloc(sizeof(int) * ARR_SIZE);

    for (int i=0; i <= ARR_SIZE; i++) {
        intArray[i] = i;
    }

    srand(time(NULL));
    int randNum = rand() % ARR_SIZE;

    printf("intArray[%d]: %d\n", randNum, intArray[randNum]);

    return 0;
}
```

- The code compiles fine without warnings
- It runs as expected as well, and thus we do not expect gdb can be applicable here

```
-> gcc -Wall main.c
juliu::Kolmogorov {~/MAE5032/week_10/valgrind }
-> ./a.out
intArray[221]: 221
```


Valgrind session

- Compile the code with `-g` is better.
- You may compile with `-O1` but the error message may be inaccurate.
- Simply put **valgrind** before the executable. `valgrind ./program <args ...>`

```
-> valgrind ./a.out
==255947== Memcheck, a memory error detector
==255947== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==255947== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright info
==255947== Command: ./a.out
==255947==
==255947== Invalid write of size 4
==255947==    at 0x40067C: main (in /work/mae-liuj/mae-5032/10-valgrind-02/a.out)
==255947== Address 0x5204fe0 is 0 bytes after a block of size 4,000 alloc'd
==255947==    at 0x4C29E33: malloc (vg_replace_malloc.c:299)
==255947==    by 0x400657: main (in /work/mae-liuj/mae-5032/10-valgrind-02/a.out)
==255947==
intArray[287]: 287
==255947==
==255947== HEAP SUMMARY:
==255947==    in use at exit: 4,000 bytes in 1 blocks
==255947== total heap usage: 1 allocs, 0 frees, 4,000 bytes allocated
==255947==
==255947== LEAK SUMMARY:
==255947==    definitely lost: 4,000 bytes in 1 blocks
==255947==    indirectly lost: 0 bytes in 0 blocks
==255947==    possibly lost: 0 bytes in 0 blocks
==255947==    still reachable: 0 bytes in 0 blocks
==255947==    suppressed: 0 bytes in 0 blocks
==255947== Rerun with --leak-check=full to see details of leaked memory
==255947==
==255947== For counts of detected and suppressed errors, rerun with: -v
==255947== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Valgrind session

- Compile the code with `-g` is better. You may compile with `-O1` but the error message may be inaccurate.
- Simply put **valgrind** before the executable.

```
--> valgrind ./a.out
==255947== Memcheck, a memory error detector
==255947== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==255947== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright info
==255947== Command: ./a.out
==255947==
==255947== Invalid write of size 4
==255947==   at 0x40067C: main (in /work/mae-liuj/mae-5032/10-valgrind-02/a.out)
==255947== Address 0x5204fe0 is 0 bytes after a block of size 4,000 alloc'd
==255947==   at 0x4C29E33: malloc (vg_replace_malloc.c:299)
==255947==   by 0x400657: main (in /work/mae-liuj/mae-5032/10-valgrind-02/a.out)
==255947==
intArray[287]: 287
==255947==
==255947== HEAP SUMMARY:
==255947==   in use at exit: 4,000 bytes in 1 blocks
==255947== total heap usage: 1 allocs, 0 frees, 4,000 bytes allocated
==255947==
==255947== LEAK SUMMARY:
==255947==   definitely lost: 4,000 bytes in 1 blocks
==255947==   indirectly lost: 0 bytes in 0 blocks
==255947==   possibly lost: 0 bytes in 0 blocks
==255947==   still reachable: 0 bytes in 0 blocks
==255947==   suppressed: 0 bytes in 0 blocks
==255947== Rerun with --leak-check=full to see details of leaked memory
==255947==
==255947== For counts of detected and suppressed errors, rerun with: -v
==255947== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

We can read the heap usage

Four categories of memory leaks

Valgrind session

- **Definitely lost** indicates blocks that are not deleted at the end of the program (return of the main function).

```
void f()  
{  
    int * y = new int[3]; // 12 bytes definitely lost  
}  
  
int main()  
{  
    f();  
    int* x = new int[10]; // 40 bytes definitely lost  
}
```

Valgrind session

- **Indirectly lost** indicates heap-allocated memory that is not directly pointed to by a program variable, but instead was only reachable through another block of memory that itself was lost (i.e. lost due to a chain of leaks).

```
struct A
{
    int* array;
};

int main()
{
    A* x = new A;           // 8 bytes definitely lost
    x->array = new int[4];   // 16 bytes indirectly lost
}
```

Valgrind session

- **Still reachable** indicates memory that is allocated on the heap, is not freed, and can still be accessed through a valid pointer at the end of the program.

```
int main()
{
    int* array = new int[3];
    std::abort(); // early abnormal termination
    // 12 bytes still reachable
    ... // maybe it is delete here
}
```

Valgrind session

- **Possibly lost** indicates memory that might be leaked, but Valgrind is not sure, because the pointer to it are ambiguous.

```
int main()  
{  
    char * p = (char*) malloc(100);  
    p += 10;  
}
```

Valgrind session

- Compile the code with `-g` is better. You may compile with `-O1` but the error message may be inaccurate.
- Simply put **valgrind** before the executable.

```
-> valgrind ./a.out
==255947== Memcheck, a memory error detector
==255947== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==255947== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright info
==255947== Command: ./a.out
==255947==
==255947== Invalid write of size 4
==255947==    at 0x40067C: main (in /work/mae-liuj/mae-5032/10-valgrind-02/a.out)
==255947== Address 0x5204fe0 is 0 bytes after a block of size 4,000 alloc'd
==255947==    at 0x4C29E33: malloc (vg_replace_malloc.c:299)
==255947==    by 0x400657: main (in /work/mae-liuj/mae-5032/10-valgrind-02/a.out)
==255947==
intArray[287]: 287
==255947==
==255947== HEAP SUMMARY:
==255947==    in use at exit: 4,000 bytes in 1 blocks
==255947== total heap usage: 1 allocs, 0 frees, 4,000 bytes allocated
==255947==
==255947== LEAK SUMMARY:
==255947==    definitely lost: 4,000 bytes in 1 blocks
==255947==    indirectly lost: 0 bytes in 0 blocks
==255947==    possibly lost: 0 bytes in 0 blocks
==255947==    still reachable: 0 bytes in 0 blocks
==255947==    suppressed: 0 bytes in 0 blocks
==255947== Rerun with --leak-check=full to see details of leaked memory
==255947==
==255947== For counts of detected and suppressed errors, rerun with: -v
==255947== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

If you cannot locate the issue, follow the instruction here

Valgrind session

Advanced flags:

`--leak-check=full` prints details for each definitely lost or possibly lost block, including where it was allocated

`--show-leak-kinds=all` to combine with `--leak-check=full` to print all leak kinds

`--track-fds=yes` list open file descriptors on exit (not closed)

`--track-origins=yes` track the origin of uninitialized values

```
valgrind --leak-check=full --show-leak-kinds=all  
        --track-fds=yes --track-origins=yes ./program <args...>
```

Track stack usage:

```
valgrind --tool=drd --show-stack-usage=yes ./program <args...>
```



```
-> valgrind --leak-check=full ./a.out
==399046== Memcheck, a memory error detector
==399046== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==399046== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright info
==399046== Command: ./a.out
==399046==
==399046== Invalid write of size 4
==399046==    at 0x40067C: main (in /work/mae-liuj/mae-5032/10-valgrind-02/a.out)
==399046== Address 0x5204fe0 is 0 bytes after a block of size 4,000 alloc'd
==399046==    at 0x4C29E33: malloc (vg_replace_malloc.c:299)
==399046==    by 0x400657: main (in /work/mae-liuj/mae-5032/10-valgrind-02/a.out)
==399046==
intArray[829]: 829
==399046==
==399046== HEAP SUMMARY:
==399046==    in use at exit: 4,000 bytes in 1 blocks
==399046== total heap usage: 1 allocs, 0 frees, 4,000 bytes allocated
==399046==
==399046== 4,000 bytes in 1 blocks are definitely lost in loss record 1 of 1
==399046==    at 0x4C29E33: malloc (vg_replace_malloc.c:299)
==399046==    by 0x400657: main (in /work/mae-liuj/mae-5032/10-valgrind-02/a.out)
==399046==
==399046== LEAK SUMMARY:
==399046==    definitely lost: 4,000 bytes in 1 blocks
==399046==    indirectly lost: 0 bytes in 0 blocks
==399046==    possibly lost: 0 bytes in 0 blocks
==399046==    still reachable: 0 bytes in 0 blocks
==399046==    suppressed: 0 bytes in 0 blocks
==399046==
==399046== For counts of detected and suppressed errors, rerun with: -v
==399046== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Tells you what type of error it is.

stack trace

process ID

instruction for more details

Summary

- Valgrind checking your program can slow down by a factor of 2 to 5. So be patient.
- Issues from Valgrind needs to be seriously addressed, especially the issues of definitely lost and indirectly lost.
- Valgrind may report something like “Warning: set address perms: large range”. It is because a large chunk of memory is being allocated. If this is your intention, then it should be fine.
- Valgrind has other tools and you should refer to their webpage for more info.

Valgrind's Tool Suite

The Valgrind distribution includes the following debugging and profiling tools:

[Memcheck](#) | [Cachegrind](#) | [Callgrind](#) | [Massif](#) | [Helgrind](#) | [DRD](#) | [DHAT](#) | [Experimental Tools](#) | [Other Tools](#)

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>