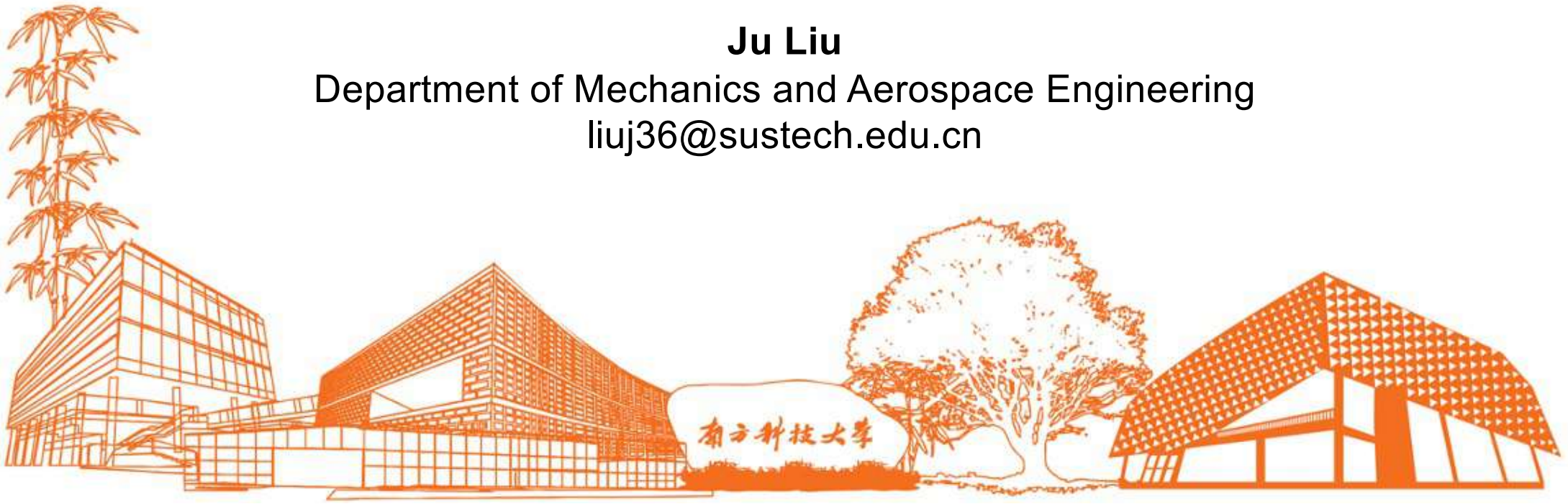
MAE 5032 High Performance Computing: Methods and Applications

Lab 5: Job control on TaiYi

Ju Liu
Department of Mechanics and Aerospace Engineering
liuj36@sustech.edu.cn



Objective

- review arrays and pointers in C
- use gdb to debug codes
- use valgrind to find memory errors

Task 1: gdb

- Obtain the source code code.c from https://github.com/ju-liu/MAE-5032-2025S/blob/main/week-11/lab/code.c
- Build the code for debugging and run it under gdb
- Set a breakpoint at main. When hitting the breakpoint, use **info locals** to see the state of the uninitialized stack variables.
- Step through the initialization statement and use **info locals** again.

Task 1: gdb

- The expression below all review to the local variable arr. First, try to figure out what the result of the expression should be, then evaluate in gdb to confirm that your understanding is correct.
  - **p *arr**
  - **p arr[1]**
  - **p &arr[1]**
  - **p arr+2**
  - **p &arr[3] – &arr[1]**
  - **p sizeof(arr)**
  - **p arr = arr+1**

Task 1: gdb

- The main function initialize ptr to arr. If you repeat the above with ptr substituted for arr, most (but not all) have the same result. The last two are different. A stack array and a pointer to it are not entirely the same. Can you explain the subtle differences?
  - **p *ptr**
  - **p ptr[1]**
  - **p &ptr[1]**
  - **p ptr+2**
  - **p &ptr[3] – &ptr[1]**
  - **p sizeof(ptr)**
  - **p ptr = ptr+1**

Task 1: gdb

- Use step to go into the call of binky. Once inside of the function, use **info args** to see values of the two parameters.

- Print all the above expressions using a and b. They will give the same result, including the last two expressions: sizeof and assignment. Can you explain?

Task 1: gdb

- Set a breakpoint on change_char and **continue** until this breakpoint is hit.

- When stopped in gdb, use **info args**. The argument shown are from the frame of reference which corresponds to the function currently executing.

- You can select the frame of reference with the frame command. Use **backtrace** to show the sequence of function calls that led to where the code is currently executing. Frames are numbered starting from 0 for the innermost frame. Try **frame 1** to select the frame outside change_char and use info locals to see the state of winky. You may use **up** and **down** to switch frames.

Task 1: gdb

- Step through change_char and examine the state before and after each line. Use **info args** to show inner frame and up and **info locals** to show what is happening in outer frame. Carefully observe the effect of each assignment statement.

Task 1: gdb

- Step through change_ptr and make the same observations. Which of the assignment statements has a persistent effect in winky and which does not? Can you explain why?

Task 2: Valgrind

- The buggy.c code has 4 planted memory errors, one that misuses stack memory and 3 misuse heap memory.

- https://github.com/ju-liu/MAE-5032-2025S/blob/main/week-11/lab/buggy.c

- When the program is invoked as ./buggy 1 name, it will copy name into a stack array declared with space for 6 characters. If name is too long, this code will write past the end of the array into the neighboring space. This kind of error is called a buffer overflow, and results in stack smashing (destroying data stored on stack next to the buffer).

Task 2: Valgrind

- Run ./a.out 1 juliu to see that the program runs correctly when the name fits.
- Try a longer name ./a.out 1 hamilton
- The detection of a stack overrun is a safety feature called stack-protector provided by the gcc compiler.
  - https://wiki.osdev.org/Stack_Smashing_Protector
- We may disable it by using the `-fno-stack-protector`
- Now rerun ./a.out 1 hamilton, we see segmentation fault.

Task 2: Valgrind

- Run ./a.out 2 juliu to see that the program runs correctly when the name fits.
- Try a longer name ./a.out 2 hamilton
- Instead of a long name overruning a stack-declared array as before, this error overruns a heap-allocated array.
- Rerun under Valgrind and read the report to see how the error is reported.

Task 2: Valgrind

- Run ./a.out 3 and ./a.out 4. They mishandle the freed heap memory.
- Review the code to see what the errors are.
- Run them under Valgrind and take not of the terminology that the Valgrind reports the uses of each.

Task 2: Valgrind

- The most frustrating errors are those that get luck and cause no observable trouble at all, but lie in wait to surprise you at the most inopportune time.

- Cultivate the habit of using Valgrind early and often in your development to detect and eradicate memory errors!