UCLA CS35L

Week 5

Wednesday

Reminders

- Assignment 4 due this Friday (5/1)
- Assignment 5 due next Friday (5/8)
- Next Friday (5/8) will be deadline to sign up for Week 10 Partners/Time Slots
    - Afterwards I will randomly assign teammates/time slots
    - I will also accept recordings for those who have time zone issues (just email me in advance)

- Survey Posted by Computer Science Department under CCLE Week 4
- Anonymous feedback for Daniel
    - https://forms.gle/tZwuMbALe825DBVn8

C Debugging

Why use Debugging Tools

- How do you usually debug code?
  - Print statements (logging)
  - Visual Debugger (Usually inside an IDE)

- Especially if your role is in the embedded or C/C++ space than debugging code with text-based tools will be valuable.
  - Valgrind – Memory Checker
  - GDB - Debugger

C Debugging - Valgrind

What is Valgrind

- Memory debugging tool – for errors and leaks primarily
- Need to compile code with debugging flag -g
- Runs your code in a lightweight VM, and emulates your environment but with profiling of memory.

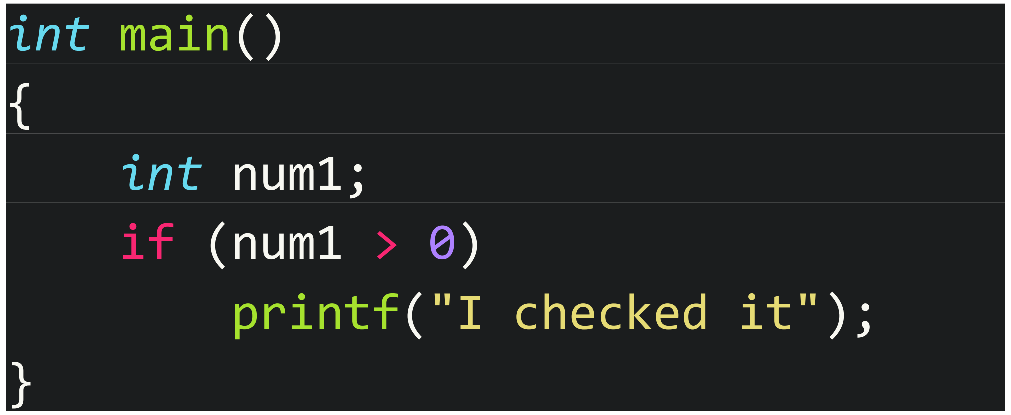
- NOTE – Examples today are with C, but also work with C++

How to use Valgrind

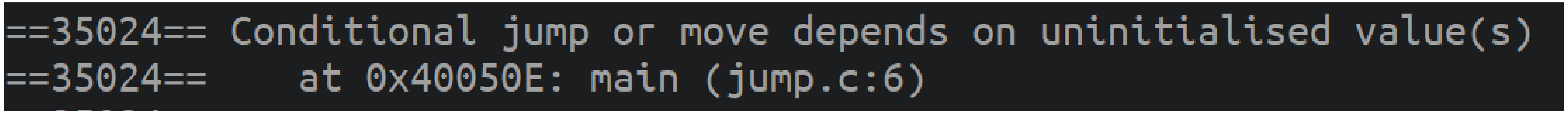
- Compile code with –g flag
  - `gcc –g source.c –o myProgram`
- Use the valgrind command, with option for more detailed leak check
  - `Valgrind --leak-check =full ./myProgram`

Valgrind – Conditional Jump on uninitialized value(s)

- Run a conditional statement on an uninitialized value



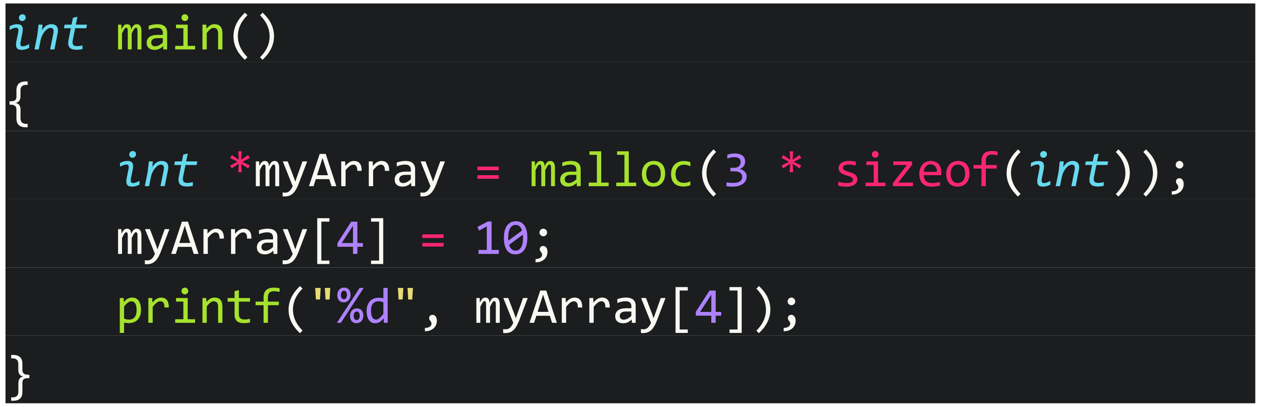
```c
int main()
{
    int num1;
    if (num1 > 0)
        printf("I checked it");
}
```



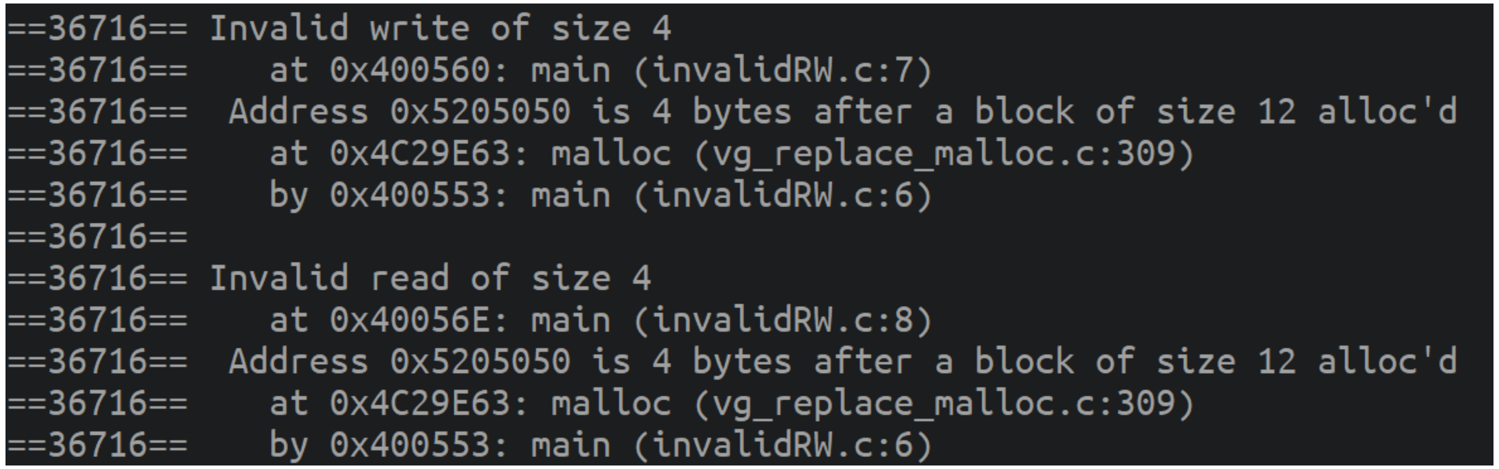
```
==35024== Conditional jump or move depends on uninitialised value(s)
==35024==    at 0x40050E: main (jump.c:6)
```

Valgrind – Invalid Read/Write

- Typically happens when accessing out-of-bounds array memory



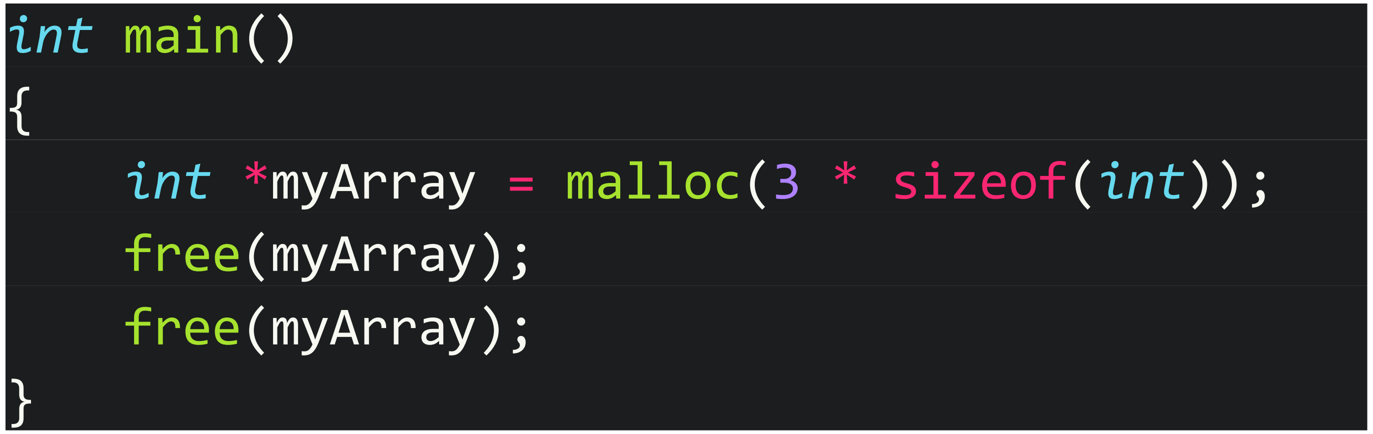
```c
int main()
{
    int *myArray = malloc(3 * sizeof(int));
    myArray[4] = 10;
    printf("%d", myArray[4]);
}
```



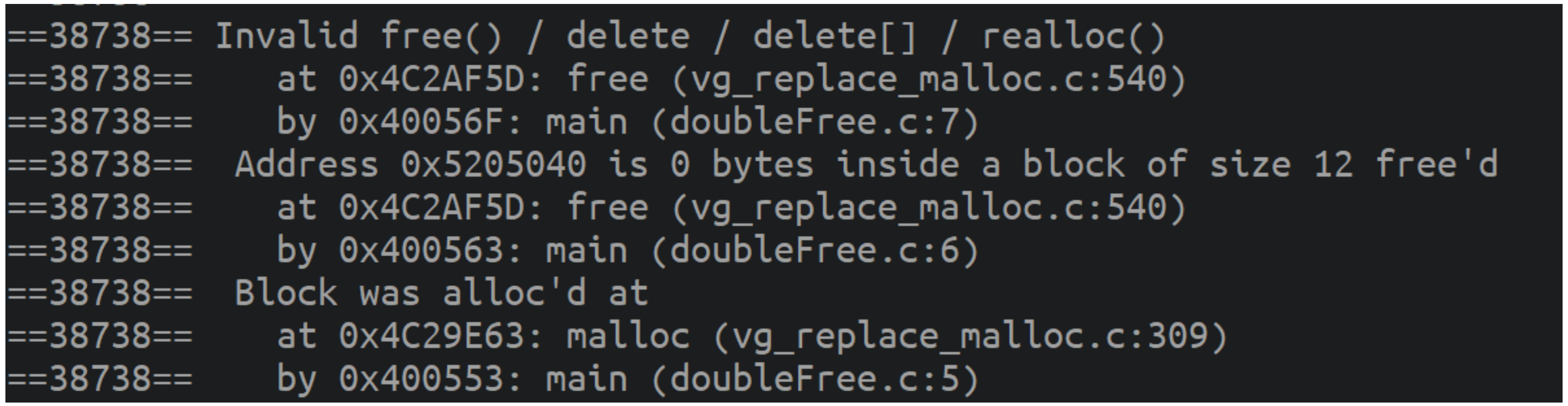
```
==36716== Invalid write of size 4
==36716==    at 0x400560: main (invalidRW.c:7)
==36716==  Address 0x5205050 is 4 bytes after a block of size 12 alloc'd
==36716==    at 0x4C29E63: malloc (vg_replace_malloc.c:309)
==36716==    by 0x400553: main (invalidRW.c:6)
==36716==
==36716== Invalid read of size 4
==36716==    at 0x40056E: main (invalidRW.c:8)
==36716==  Address 0x5205050 is 4 bytes after a block of size 12 alloc'd
==36716==    at 0x4C29E63: malloc (vg_replace_malloc.c:309)
==36716==    by 0x400553: main (invalidRW.c:6)
```

Valgrind – Invalid free

- Typically happens when you try to free() the same thing twice



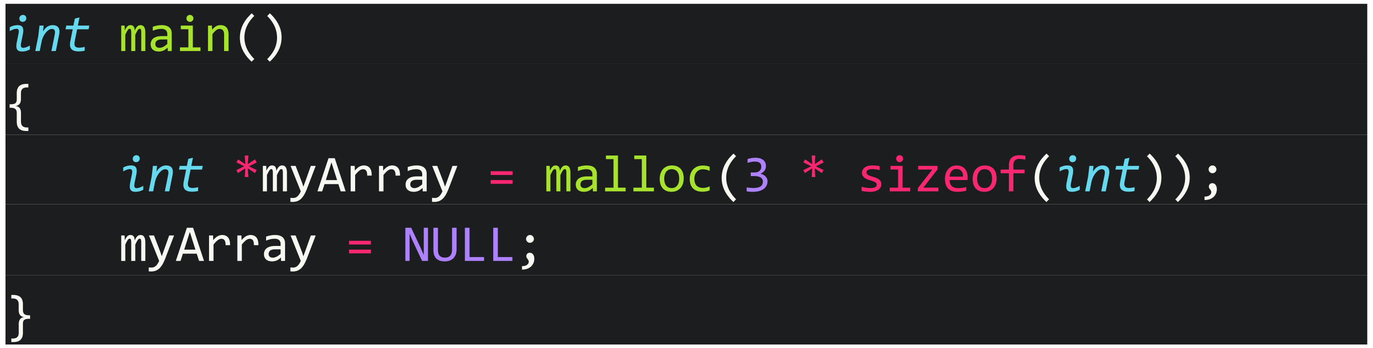
```c
int main()
{
    int *myArray = malloc(3 * sizeof(int));
    free(myArray);
    free(myArray);
}
```



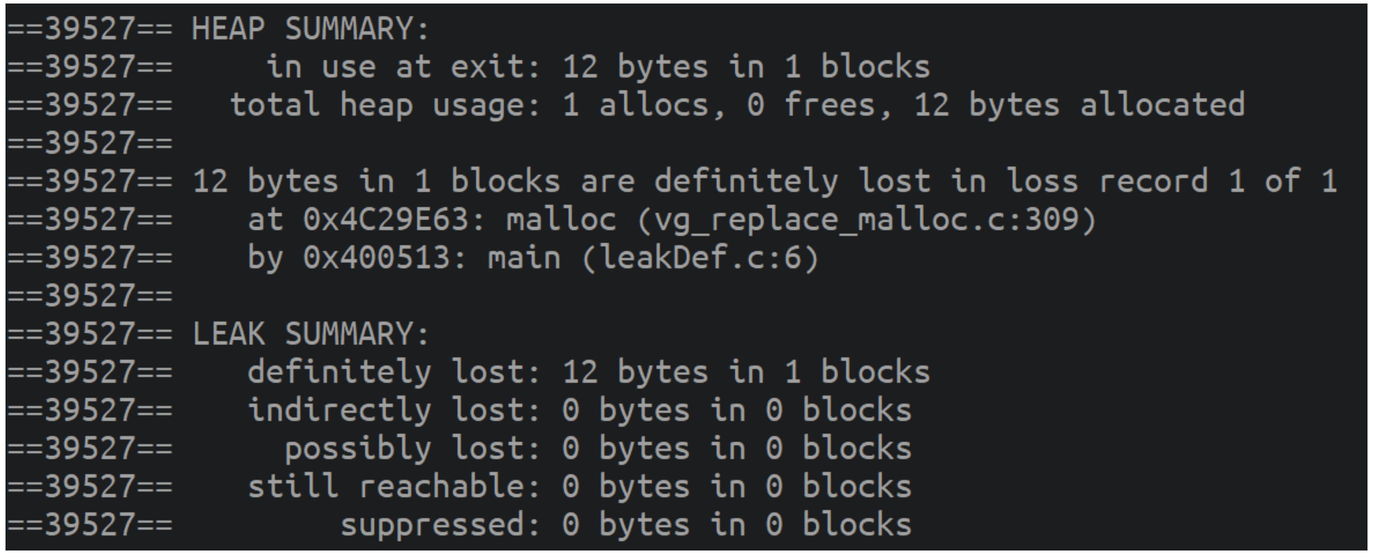
```
==38738== Invalid free() / delete / delete[] / realloc()
==38738==    at 0x4C2AF5D: free (vg_replace_malloc.c:540)
==38738==    by 0x40056F: main (doubleFree.c:7)
==38738==  Address 0x5205040 is 0 bytes inside a block of size 12 free'd
==38738==    at 0x4C2AF5D: free (vg_replace_malloc.c:540)
==38738==    by 0x400563: main (doubleFree.c:6)
==38738==  Block was alloc'd at
==38738==    at 0x4C29E63: malloc (vg_replace_malloc.c:309)
==38738==    by 0x400553: main (doubleFree.c:5)
```

Valgrind – Leak Detection

- You lost your pointer to a dynamic memory location



```c
int main()
{
    int *myArray = malloc(3 * sizeof(int));
    myArray = NULL;
}
```



```
==39527== HEAP SUMMARY:
==39527==     in use at exit: 12 bytes in 1 blocks
==39527==   total heap usage: 1 allocs, 0 frees, 12 bytes allocated
==39527==
==39527== 12 bytes in 1 blocks are definitely lost in loss record 1 of 1
==39527==    at 0x4C29E63: malloc (vg_replace_malloc.c:309)
==39527==    by 0x400513: main (leakDef.c:6)
==39527==
==39527== LEAK SUMMARY:
==39527==    definitely lost: 12 bytes in 1 blocks
==39527==    indirectly lost: 0 bytes in 0 blocks
==39527==      possibly lost: 0 bytes in 0 blocks
==39527==    still reachable: 0 bytes in 0 blocks
==39527==         suppressed: 0 bytes in 0 blocks
```

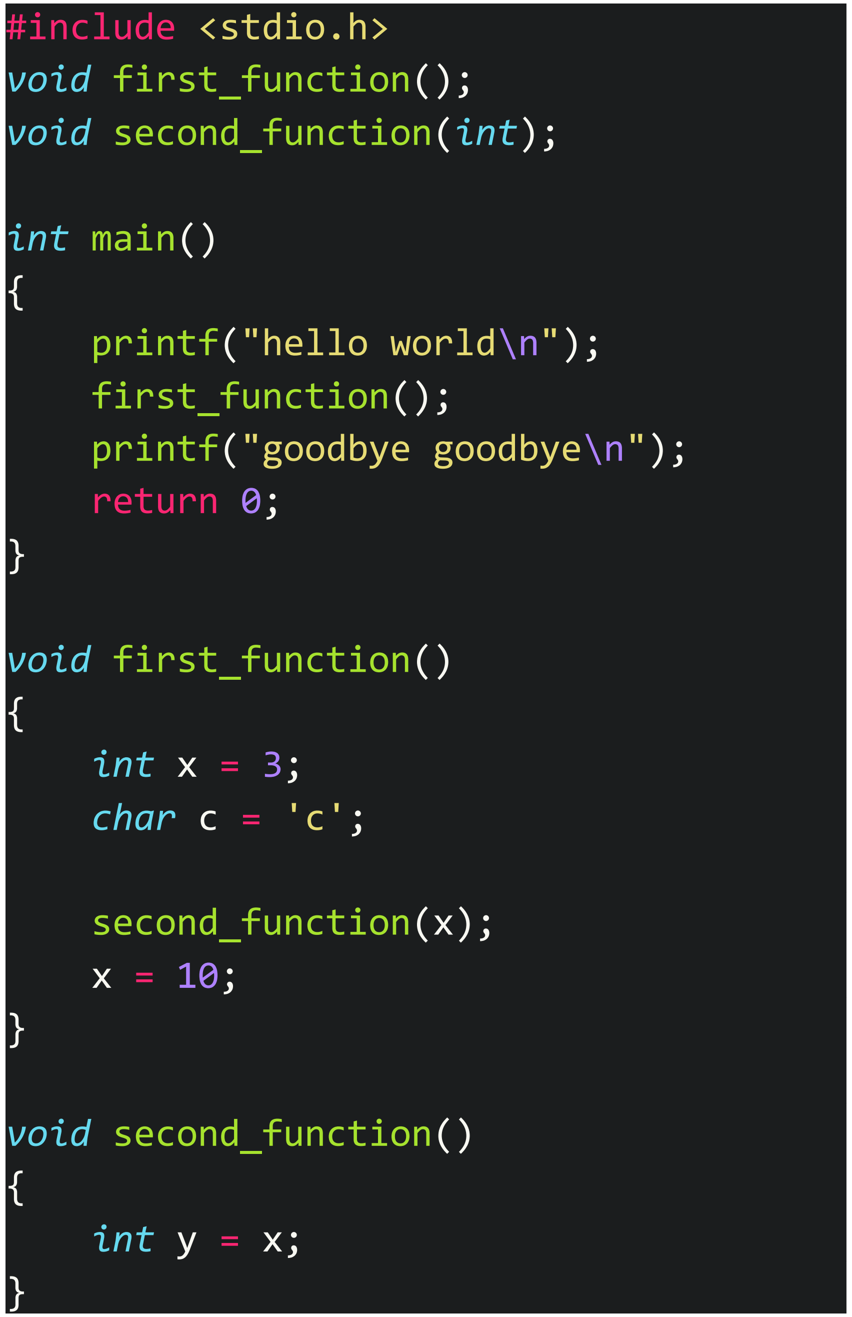
C Debugging - GDB

What is GDB

- A Command-Line Debugger
- Gives many of the similar capabilities that a visual debugger like Visual Studio will, but sometimes you may only have access to the command-line.
  - Also can analyze Core Dumps for crashes

- Remember to compile with -g

Example Program



```c
#include <stdio.h>
void first_function();
void second_function(int);

int main()
{
    printf("hello world\n");
    first_function();
    printf("goodbye goodbye\n");
    return 0;
}

void first_function()
{
    int x = 3;
    char c = 'c';

    second_function(x);
    x = 10;
}

void second_function()
{
    int y = x;
}
```

Start GDB

- Two options
  - `gdb [yourExec]`
  - `gdb` then `file <yourExec>`

- After file is loaded use
  - `run [arguments]`
- Or to load from stdin
  - `run < fileForSTDIN`

Breakpoints

- Breakpoints are places where you pause automatic code execution, and give you more manual control
- Create breakpoints
  - `break [filename:] line_number` – set a breakpoint at that line
  - `break function` – set a breakpoint at the first line of the function
  - `break` – set a breakpoint on the next line
- Can add conditionals
  - `break my_func if …`
- `info b –` list all breakpoint information

Changing Breakpoints

- `delete [bp_number | range]` – deletes the specified breakpoints
- `disable [bp_number | range]` – disables the specified breakpoints
- `enable [bp_number | range]` – enables the specified breakpoints
- `ignore bp_number iterations` – Pass over a specific breakpoint for a specific number of times

Controlling Program Execution

- Once you hit a breakpoint, it’s up to you how you want to proceed

`continue` – resume automatic execution until the next breakpoint

`step [n]` – step to next [n] lines of code. Will “step” into any functions

`next [n]` – Similar to step, but does not go inside functions

Observing Variables

- You can check the value of any variable at any time with print
  - `print var`
- Or you can set a watchpoint that pauses execution when var changes
  - `watch var`
- Or you can set a watchpoint for when var is read
  - `rwatch var`

Using GDB to check the Stack

- `info frame`
  - Displays info about the stack frame, including its return address and saved register values
- `info local`
  - List the local variables of the function corresponding to the stack frame, and their current values
- `info args`
  - List the argument variables for the corresponding function call

- NOTE – More relevant in classes like CS33, you probably won’t use this much in this class