

# CS 211 : Thurs 01/18 (lecture 05)



*Prof. Hummel*  
(he/him)

- Topics: valgrind, gdb, function design

## January 2024

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

[www.a-printable-calendar.com](http://www.a-printable-calendar.com)

## Notes:

- *Lecture slides available on Canvas*
- ***Project 02** due Friday @ 11:59pm, may be submitted up to 48 hours late. Gradescope open for submissions, test files posted*
- *Posted a **video** on using valgrind and gdb*
- *There will be a **HW 03** due Tuesday*



Northwestern  
University

## Question #1

1) *How many pointers are accessed in this line of code?*

```
stmt->types.assignment->rhs->types.expr->rhs->element->element_value
```

**A) 1**

**B) 2**

**C) 3**

**D) 4**

**E) 5 or more**

## Question #2

2) *After a series of if statements, this code can be safely executed. Which case is this code designed for?*

```
stmt->types.assignment->rhs->types.expr->rhs->element->element_value
```

**A) *print(123)***

**B) *s = input("enter something> ")***

**C) *x = 123***

**D) *x = a \* b***

**E) *both C and D***

# Function design

- In C, nearly all functions return something...

```
int main()
{
    FILE* input;
    int    return1, return2;
    int    value;

    return1 = printf("enter an integer> ");
    if (return1 != 18) ...

    return2 = scanf("%d", &value);
    if (return2 != 1) ...

    input = fopen("test.py", "r");
    if (input == NULL) ...

    .
    .
    .
}
```

# Returning success *\*and\** data

- Approach #1: return a pointer to the data
  - *Returning NULL if the call failed...*

```
int main()
{
    FILE* input;
    int    return1, return2;
    int    value;

    return1 = printf("enter an integer> ");
    if (return1 != 18) ...

    return2 = scanf("%d", &value);
    if (return2 != 1) ...

    input = fopen("test.py", "r");
    if (input == NULL) ...

    .
    .
    .

    fclose(input);
```

The function has to **malloc( )** memory to hold the data that is returned...

And the caller has to free that memory when done...

## Question #3

3) Suppose I want to write a function that returns a Token.  
Which one is correct?

```
struct Token {  
    int ID;  
    int line;  
    int col;  
};
```

(A)

```
struct Token* myfunction(int ID, int L, int C) {  
    struct Token* T;  
    T = (struct Token*) malloc(sizeof(struct Token));  
    T->ID = ID;  
    T->line = L;  
    T->col = C;  
    return T;  
}
```

(B)

```
struct Token* myfunction(int ID, int L, int C) {  
    struct Token* T;  
    T = (struct Token*) malloc(sizeof(struct Token*));  
    T->ID = ID;  
    T->line = L;  
    T->col = C;  
    return T;  
}
```

(C)

```
struct Token* myfunction(int ID, int L, int C) {  
    struct Token* T;  
    T = (struct Token*) malloc(sizeof(T));  
    T->ID = ID;  
    T->line = L;  
    T->col = C;  
    return T;  
}
```

# Returning success \*and\* data

- Approach #2: use return value + pointer parameter

```
int main()
{
    FILE* input;
    int    return1, return2;
    int    value;

    return1 = printf("enter an integer> ");
    if (return1 != 18) ...

    return2 = scanf("%d", &value);
    if (return2 != 1) ...

    input = fopen("test.py", "r");
    if (input == NULL) ...

    .
    .
    .
}
```

*Caller needs to pass parameter properly...*

## Question #4

4) Write a function that computes the sum of an array, returning false if empty and true if the array has data. Which is correct?

(A)

```
bool array_sum(int* A, int N, int* sum)
{
    if (N <= 0) return false;

    sum = 0;
    for (int i=0; i<N; i++) { sum += A[i]; }
    return true;
}
```

(B)

```
bool array_sum(int* A, int N, int* sum)
{
    if (N <= 0) return false;

    *sum = 0;
    for (int i=0; i<N; i++) { *sum += A[i]; }
    return true;
}
```

(C)

```
bool array_sum(int* A, int N, int* sum)
{
    if (N <= 0) return false;

    *sum = 0;
    for (int i=0; i<N; i++) { *sum += *A; }
    return true;
}
```



# How about this approach?

- Approach #3: use a struct...

```
int main()
{
    struct Answer A;

    A = myfunction(...);

    if (A.success)
        printf("Answer=%d\n", A.result);
}
```

```
struct Answer {
    bool success;
    int  result;
};
```

- Why is this approach not used in C?

- A) *Does not generalize easily*
- B) *Requires more memory*
- C) *Slower execution*
- D) *Creating structs is tedious*
- E) *All of the above*



**nuPython**

# Function design

- **The project provides some design choices...**
- **Example:**
- **How does an assignment statement execute?**

# Implementation?

- The implementation should reflect this as closely as possible...

$$X = a * b$$

1. read a

2. read b

3. multiply

4. write result





# C

Programming

# What if my program crashes?

- My program crashes, now what?

```

>_ Console x Shell x +
  ▶ Run

❖ gcc -std=c11 -g -Wall -lm main.c scanner.c compiler.o -Wno-
❖ ./a.out
Enter nuPython file (press ENTER to input from keyboard)>
test06.py
**parsing successful
**building AST...
**starting execution...

TEST CASE: test06.py

enter any string> 123
123
exit status -1
❖ █

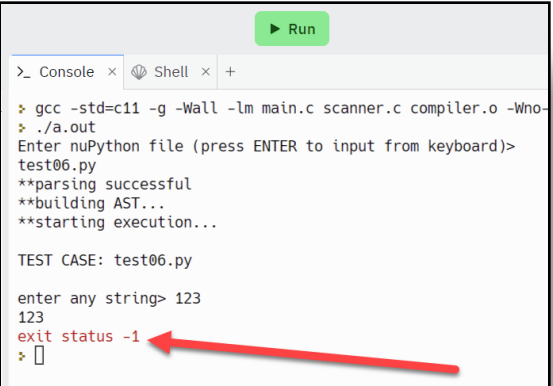
```



# Approaches

- **Debugging approaches...**

- *Print debugging*
- *Change the input / test case --- what input triggers the error? What code handles that input?*
- *Comment out code and run again, repeat until error goes away...*



```
>_ Console x Shell x +
> gcc -std=c11 -g -Wall -lm main.c scanner.c compiler.o -Who-
> ./a.out
Enter nuPython file (press ENTER to input from keyboard)>
test06.py
**parsing successful
**building AST...
**starting execution...

TEST CASE: test06.py

enter any string> 123
123
exit status -1
> 
```

# Tools

- **Tools are often the quickest way to locate an error**
  - `valgrind ./a.out`
  - `gdb ./a.out`

*Posted a 5-minute video on Canvas...*



# valgrind

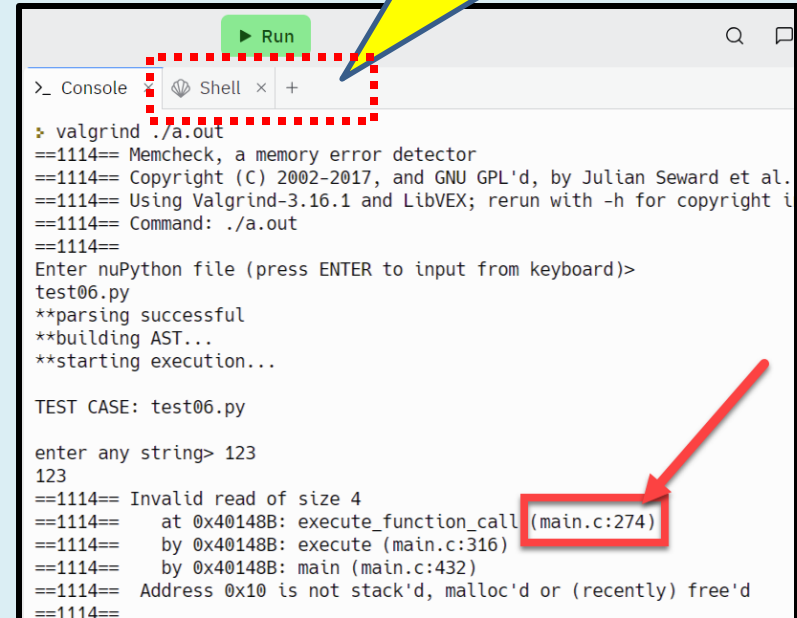
- Use valgrind to find:

- *Memory errors*

- *Memory leaks*

- valgrind --tool=memcheck --leak-check=full ./a.out

*Run in Shell (replit) or  
terminal window (VS Code)*



```
>_ Console x Shell x +
* valgrind ./a.out
==1114== Memcheck, a memory error detector
==1114== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1114== Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright i
==1114== Command: ./a.out
==1114==
Enter nuPython file (press ENTER to input from keyboard)>
test06.py
**parsing successful
**building AST...
**starting execution...

TEST CASE: test06.py

enter any string> 123
123
==1114== Invalid read of size 4
==1114==    at 0x40148B: execute_function_call (main.c:274)
==1114==    by 0x40148B: execute (main.c:316)
==1114==    by 0x40148B: main (main.c:432)
==1114== Address 0x10 is not stack'd, malloc'd or (recently) free'd
==1114==
```

# **gdb**

- Use gdb to interactively explore what your program is doing
  - *You can set breakpoints to stop at any line*
  - *You can execute your program line by line*
  - *You can print the contents of memory*
  - **gdb ./a.out**

```
hummel> gdb ./a.out
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./a.out...
(gdb) break execute
Breakpoint 1 at 0x1b14: file execute.c, line 301.
(gdb) run
Starting program: /mnt/c/classes/cs211/05 Thurs 01-18 valgrind, gdb, function design/Demo/nuPython/a.out
nuPython input (enter $ when you're done)
x = 123
$
**no syntax errors...
**building program graph...
**PROGRAM GRAPH PRINT**
1: x = 123
2: $
**END PRINT**
**executing...

Breakpoint 1, execute (program=0x4ffffdf80, memory=0x55555555a1d0 <__libc_csu_init>) at execute.c:301
301 {
(gdb) next
302     struct STMT* stmt = program;
(gdb) next
307     while (stmt != NULL) {
(gdb) p stmt
$1 = (struct STMT *) 0x555555560ac0
(gdb) p stmt->stmt_type
$2 = 0
(gdb) p stmt->types.assignment->var_name
$3 = 0x555555560c00 "x"
(gdb) |
```

# Debugging with gdb

- **Commands:**

- ***break***            *set breakpoint using line # or function name*
- ***run***                *run program*
- ***continue***        *continue execution after stopping*
- ***p***                    *print contents of memory*
- ***next***                *execute next line of program and stop*
- ***step***                *step into a function call so you can execute it*
- ***where***              *where am I in the program (function, line #)*
- ***ENTER***            *repeat last command*
- ***quit***                *stop debugging*

# What should I be working on?

*Project 02 is due Friday night...*

*Watch for release of **HW 03** and **Project 03**...*

