

__init__ Monty

BY ME.NAME == CHRIS

LIFO v FIFO (context != GAAP)

- ▶ FIFO == First In First Out E.G. an amusement park line
- ▶ LIFO == Last In First Out E.G. conventional spring loaded magazine

Similar to the acronyms themselves, the usage of these methodologies will be largely context dependent:

Suppose you want to implement an undo feature for a word processor like Word, which method would be best?

Suppose you're building a customer callback system, which method would be best then?

STACK v HEAP – Memory Man

1. **Memory Allocation:** Heap is a dynamically allocated memory that is used for storing data that persists beyond the scope of the current function. Stack, on the other hand, is a statically allocated memory used for storing data that is local to the current function.
2. **Memory Management:** Heap memory is managed manually by the programmer, which means that the programmer is responsible for allocating and deallocating memory when required. Stack memory, on the other hand, is managed automatically by the operating system.
3. **Memory Access:** Heap memory can be accessed by any part of the program as long as a valid pointer to that memory is available. Stack memory can only be accessed by the function that allocated it.
4. **Memory Size:** Heap memory is limited only by the amount of available memory on the computer. Stack memory, on the other hand, is limited in size and can overflow if too much data is pushed onto it.
5. **Performance:** Heap memory allocation and deallocation are slower than stack memory allocation and deallocation. This is because heap memory involves more complex operations such as searching for free memory blocks and updating pointers.

valgrind && exit(STATUS)

←GOOD RUN
BAD RUN ----->

Install valgrind:
“**sudo apt-get install valgrind**”

Use valgrind:
“**valgrind {{path to exec}} ...**”

←GOOD EXIT (0)
BAD EXIT (1)-->

```
test - grazingtatanka [WSL: Ubuntu] - Visual Studio Code
test M X
inactiverepos > holbertonschool-monty > test
1 push 1
2 push 2
3 push 3
4 push 4
5 push 0
6 push 110
7 push 0
8 push 108
9 push 111
10 push 111
11 push 104
12 push 99
13 push 83
14 pstr

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE
● grazingtatanka@PC:~/inactiverepos/holbertonschool-monty$ vgrind ./monty test
==350== Memcheck, a memory error detector
==350== Copyright (c) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==350== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==350== Command: ./monty test
==350==
School
==350==
==350== HEAP SUMMARY:
==350==   in use at exit: 0 bytes in 0 blocks
==350== total heap usage: 17 allocs, 17 frees, 6,024 bytes allocated
==350==
==350== All heap blocks were freed -- no leaks are possible
==350==
==350== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
○ grazingtatanka@PC:~/inactiverepos/holbertonschool-monty$
```

```
● grazingtatanka@PC:~/inactiverepos/holbertonschool-monty$ ./monty test
School
○ grazingtatanka@PC:~/inactiverepos/holbertonschool-monty$ $?
1: command not found
○ grazingtatanka@PC:~/inactiverepos/holbertonschool-monty$
```

```
test - grazingtatanka [WSL: Ubuntu] - Visual Studio Code
test M X
inactiverepos > holbertonschool-monty > test
1 push 1
2 push 2
3 push 3
4 push 4
5 push 0
6 push 110
7 push 0
8 push 108
9 push 108
10 push 111
11 push 111
12 push 104
13 push 99
14 push 83
15 pstr

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE
○ grazingtatanka@PC:~/inactiverepos/holbertonschool-monty$ vgrind ./monty test
==592== Memcheck, a memory error detector
==592== Copyright (c) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==592== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==592== Command: ./monty test
==592==
L8: usage: push integer
==592==
==592== HEAP SUMMARY:
==592==   in use at exit: 0 bytes in 0 blocks
==592== total heap usage: 10 allocs, 10 frees, 4,856 bytes allocated
==592==
==592== All heap blocks were freed -- no leaks are possible
==592==
==592== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
○ grazingtatanka@PC:~/inactiverepos/holbertonschool-monty$
```

```
○ grazingtatanka@PC:~/inactiverepos/holbertonschool-monty$ ./monty test
L8: usage: push integer
○ grazingtatanka@PC:~/inactiverepos/holbertonschool-monty$ $?
1: command not found
○ grazingtatanka@PC:~/inactiverepos/holbertonschool-monty$
```

Why you don't GENERALLY want to use globally scoped variables in C

- ▶ Performance – slower access and more memory
- ▶ Maintenance – changing one component may affect many
- ▶ Name Clashes – since accessible anywhere, can conflict
- ▶ Debugging – accessible anywhere, so hard to find problem source
- ▶ Security – if everywhere, odds of being in affected area is 100%

SOLUTION SET && EXAMPLE OF GLOBAL STRUCT

- ▶ Since the interpreter we're building will already be lightweight from a resource perspective, not have ongoing maintenance concerns, likely only be comprised of a handful of files, and at comically low risk of exploit attempts – use global struct! TY @Autumn

```
monty.h - grazingtatanka [WSL: Ubuntu] - Visual Studio Code
C monty.h x
inactiverepos > holbertonschool-monty > C monty.h > [daedalus]
43 } instruction_t;
44
45 /**
46  * struct global_s - global struct
47  * @op_code: the opcode
48  * @op_arg: associated argument if applicable
49  * @op_mode: operation mode
50  * @op_line: line of inbound file
51  * @line_ref: pointer to line
52  * @file_ref: pointer to FILE
53  * Description: The Way
54  * for stack, queues, LIFO, FIFO
55  */
56 typedef struct global_s
57 {
58     char *op_code;
59     char *op_arg;
60     unsigned int op_mode;
61     unsigned int op_line;
62     char *line_ref;
63     FILE *file_ref;
64 } global_t;
65
66 extern struct global_s daedalus;
67
```

cmp(interpreted, compiled)

Compiled (e.g. C, Java)

- ▶ Code executed by CPU
- ▶ Debug < (worse)
- ▶ Dev_time > (worse)
- ▶ Speed > (better)
- ▶ Portability < (worse)

Interpreted (e.g. python, JS)

- ▶ Code executed by interpreter
- ▶ Debug > (better)
- ▶ Dev_time < (better)
- ▶ Speed < (worse)
- ▶ Portability > (better)

Handle EOF (End-Of-File)

- ▶ In Unix-like operating systems, when the end of file (EOF) is reached on a file or input stream, the `read()` system call returns zero to indicate the end of the file.
- ▶ **!WARNING! WITH `GETLINE()` THE EOF CONDITION WILL BE INDICATED BY AN (INT) -1 RETURN; SEE FUNCTION VS. SYSTEM CALLS**

```
while (getline(&line_buff, &n, inbound_file) != -1)
```

- ▶ How long will this loop continue? What causes it to stop?

GETLINE

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

```
ssize_t getline(char **lineptr, size_t *n, FILE *stream);
```

STRtok

```
#include <string.h>
```

```
char *strtok(char *str, const char *delim);
```

FPRINTF

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

```
int fprintf(FILE *stream, const char *format, ...);
```

FOPEN

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

```
FILE *fopen(const char *pathname, const char *mode);
```