__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:

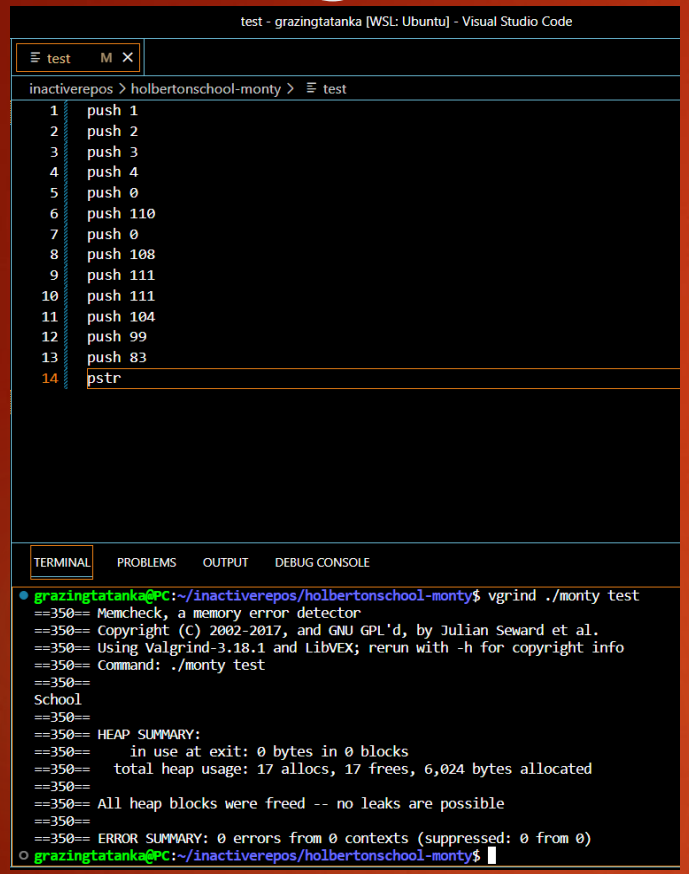
Suppose you want to implement an undo feature for a word processor like Word, which method would be best?

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.

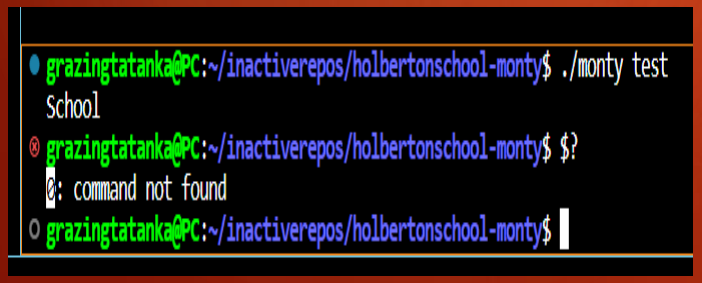
valgrind && exit(STATUS)



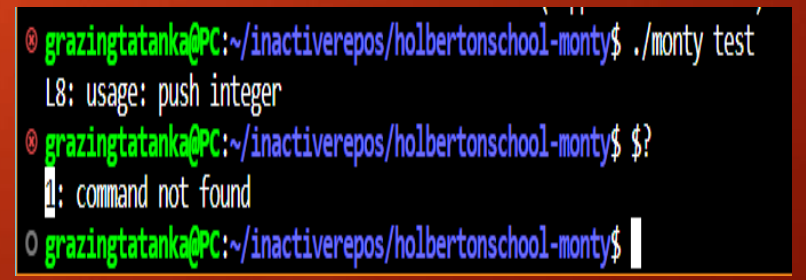
←GOOD RUN
BAD RUN ----->

Install valgrind:
“sudo apt-get install valgrind”

Use valgrind:
“valgrind {{path to exec}} …”



←GOOD EXIT (0)
BAD EXIT (1)-->

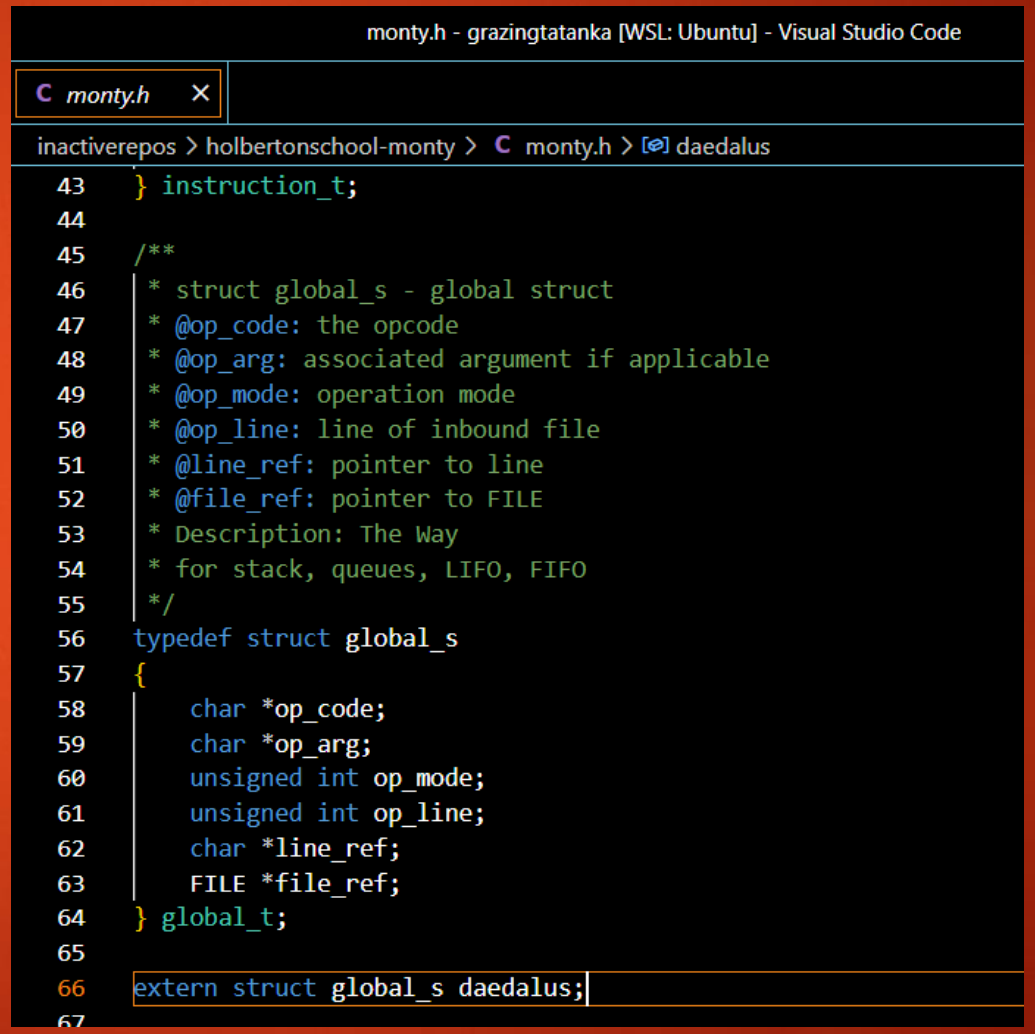


Why you don't GENERALLY want to use globally scoped variables in C

- 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



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);
```