Heap, Makefiles, Debugging Tools

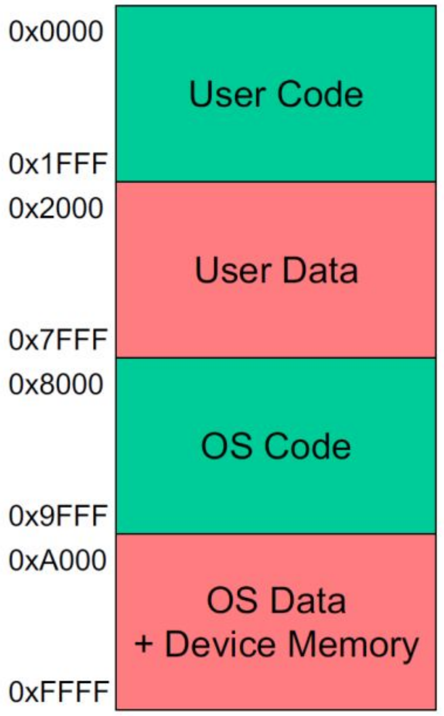
CIS 2400 Recitation 8

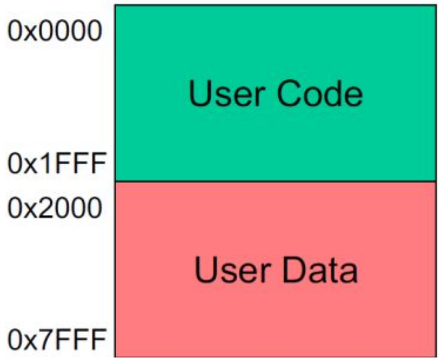
Recitation Outline

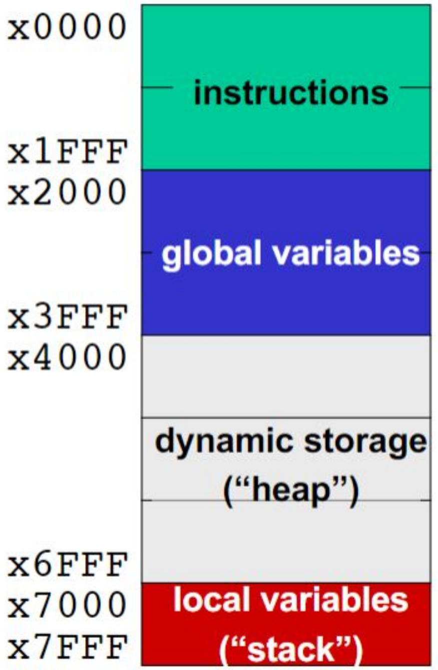
- Heap
  - Dynamic memory allocation
  - Structs
- Makefiles
- [For reference] Debugging tools cheat sheets
  - GDB
  - Valgrind

Heap

LC4 Memory Breakdown: User Space







Static vs. Dynamic Memory Allocation

- Static = Compile time
  - Allocate a chunk of memory to a variable of known size

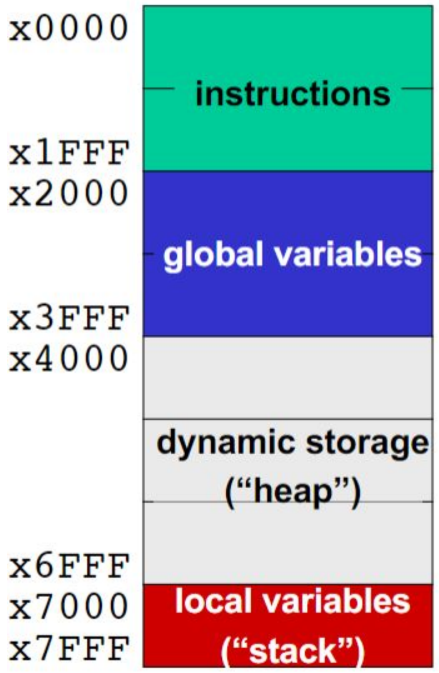
```
#define MAX_LINE_LENGTH = 100
char user_input[MAX_LINE_LENGTH]
```

- Dynamic = Runtime
  - Allocate a chunk of memory to a variable of unknown size at compile time (e.g. struct, array of unknown length)

```
#define NODE_STRUCT_SIZE = ???
// how do I allocate :((
```

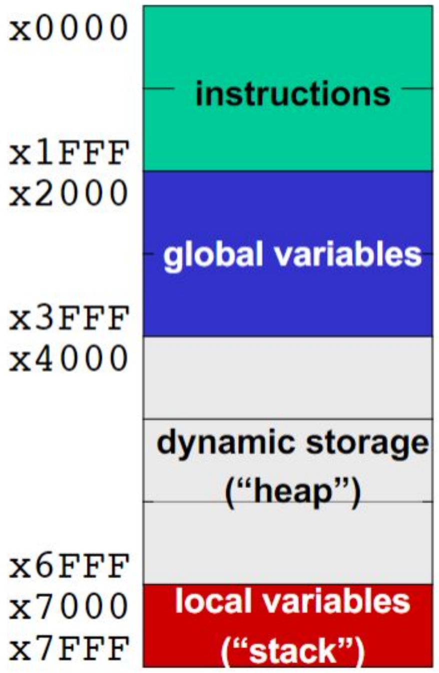
Dynamic Allocation - Heap

- Dynamic allocation is completely managed by the programmer, since the compiler does not have enough information
- Dynamically allocated variables live in the heap, where they persist until the user “frees” them
- Note that statically allocated variables will either be global variables (has their own space) or local variables (live in the stack, more on that later)



Dynamic Allocation - How to interact with the heap?

- Two functions
  - void *malloc(size_t size)
    - Gives a pointer to (address of) heap region of size size
  - void free(void *ptr)
    - Return the region pointed to by ptr to the heap (now free to use by other variables!)



Malloc - how to get the size?

- void *malloc(size_t size)
- Variable size can be obtained using the sizeof() operator
- The sizeof operator can be applied to a variable or a type and it returns the size of that object in bytes
- Example
  - sizeof(int) - returns the size of integer
    - struct deque_struct dq
    - sizeof(dq) - returns the size of the struct dq (based on the struct fields sizes, etc)

Allocation Example

Recall strings from R7.

Static allocation:

```
char str[8];
```

Using malloc described in the previous slides, how do I allocate the same amount of space on the heap?

```
char* str = malloc(sizeof(char) * 8)
```

Structs

Struct: C datatype that contains a set of fields

- Similar to a Java class, but with no methods or constructors

Struct Example – Declaration

```
struct ll_node {
    int value;
    struct ll_node *next;
    struct ll_node *prev;
};
```

- **`struct`** tells us that we are creating a new structured data type

Struct Example – Creating, Getting/Setting Fields 1

```
struct ll_node {
    int value;
    struct ll_node *next;
    struct ll_node *prev;
};
```

Getting/setting field values

→ Use **dot notation** for structs themselves

```
struct ll_node n1;
struct ll_node n2;
n1.value = 1;
n2.value = 2;
n1.next = &n2;
n2.prev = &n1;
printf(“%d -> %d”, n1.value,
*(n1.next).value);
```

Struct Example – Creating, Getting/Setting Fields 2

```
struct ll_node {
    int value;
    struct ll_node *next;
    struct ll_node *prev;
};
```

Getting/setting field values

→ Use **arrow notation** for pointers

Syntactic sugar for

`*(mystruct).value`

```
struct ll_node *n1 =
malloc(sizeof(struct ll_node));
struct ll_node *n2 =
malloc(sizeof(struct ll_node));
n1->value = 1;
n2->value = 2;
n1->next = n2;
n2->prev = n1;
printf("%d -> %d", n1->value,
n1->next->value);
```

Struct Example – Alias Declaration

```
typedef struct ll_node {
    int value;
    struct ll_node *next;
    struct ll_node *prev;
} Node;
// OR: typedef struct ll_node Node
```

- `typedef` creates an alias for the struct
- Lets us drop the **struct** keyword in typing
- **What is the alias above?**

Struct Example – Creating, Getting/Setting Fields with Alias

```
typedef struct ll_node {
    int value;
    struct ll_node *next;
    struct ll_node *prev;
} Node;
```

Getting/setting field values

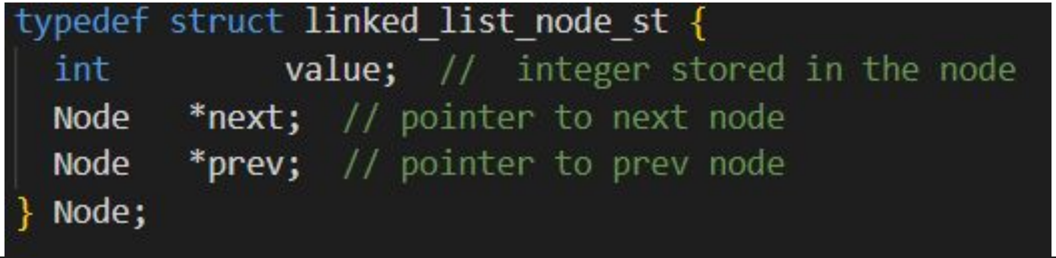
→ Use **arrow notation**

Syntactic sugar for

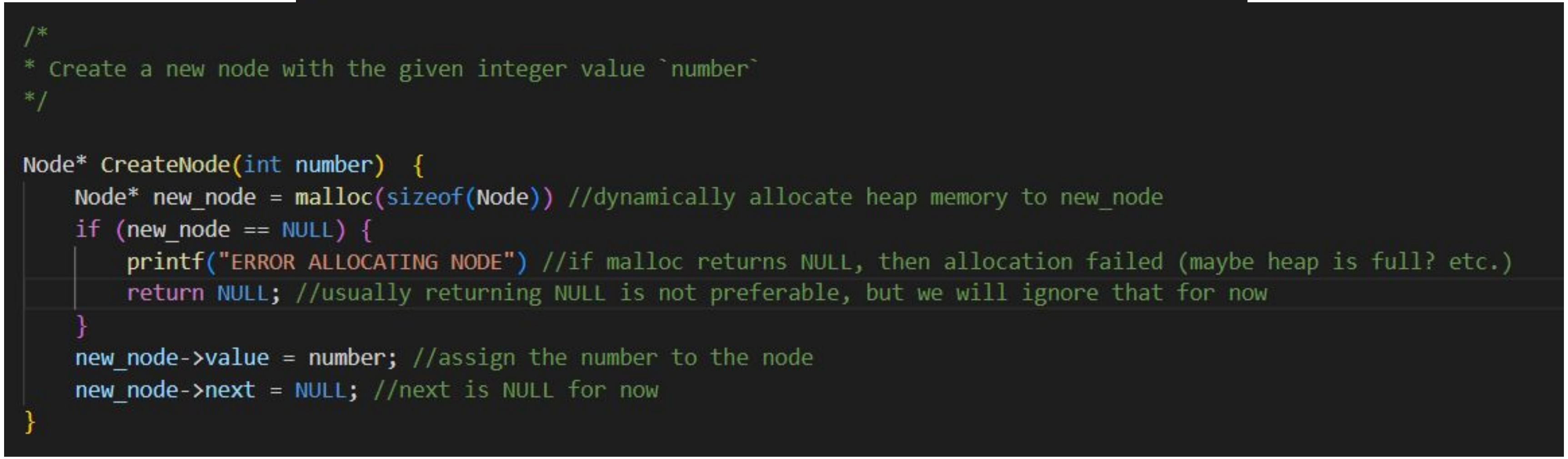
`*(mystruct).value`

```
Node *n1 = malloc(sizeof(Node));
Node *n2 = malloc(sizeof(Node));
n1->value = 1;
n1->next = n2;
n2->value = 2;
n2->prev = n1;
printf("%d -> %d", n1->value,
n1->next->value);
```

Dynamic Memory Allocation Example

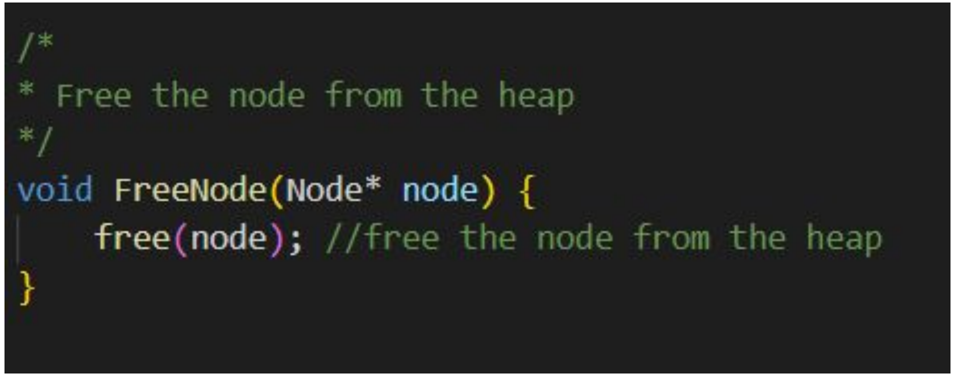


```c
typedef struct linked_list_node_st {
  int         value;  //  integer stored in the node
  Node    *next;  // pointer to next node
  Node    *prev;  // pointer to prev node
} Node;
```



```c
/*
* Create a new node with the given integer value `number`
*/

Node* CreateNode(int number)  {
    Node* new_node = malloc(sizeof(Node)) //dynamically allocate heap memory to new_node
    if (new_node == NULL) {
        printf("ERROR ALLOCATING NODE") //if malloc returns NULL, then allocation failed (maybe heap is full? etc.)
        return NULL; //usually returning NULL is not preferable, but we will ignore that for now
    }
    new_node->value = number; //assign the number to the node
    new_node->next = NULL; //next is NULL for now
}
```

Dynamic Memory Allocation Example



```c
/*
* Free the node from the heap
*/
void FreeNode(Node* node) {
    free(node); //free the node from the heap
}
```

What if I don’t free?

MEMORY LEAKS

Exercise

In a company, there are multiple teams. Teams have 1 manager and several employees, all with names. Employees may have desk neighbors to their left and/or right. For bigger teams, employees may not all be sitting in a row.

1. Create `struct`s to represent the team and each of its parts.
2. Then, write functions to
   a. Create an empty team (knowing the manager)
   b. Add an employee to a team (knowing who their desk neighbors are)
   c. Remove an employee
   d. Remove an entire team

Remember, we don’t want memory leaks!

Makefile

Why do Makefiles exist?

- Makefiles are used to help decide which parts of a large program need to be recompiled. In the vast majority of cases, C or C++ files are compiled.
- Other languages typically have their own tools that serve a similar purpose as Make.
- Make can also be used beyond compilation too, when you need a series of instructions to run depending on what files have changed.

Makefile Syntax

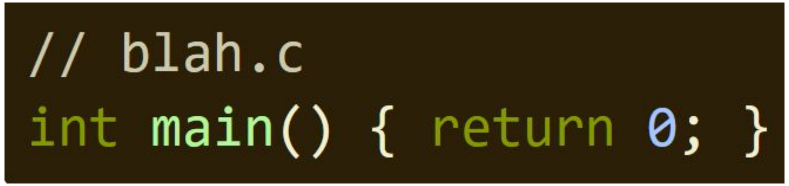
A Makefile consists of a set of rules. A rule generally looks like this:

```
targets: prerequisites
	command
	command
	command
```

- The targets are file names, separated by spaces. Typically, there is only one per rule.
- The commands are a series of steps typically used to make the target(s). These need to start with a tab character, not spaces.
- The prerequisites are also file names, separated by spaces. These files need to exist before the commands for the target are run. These are also called dependencies

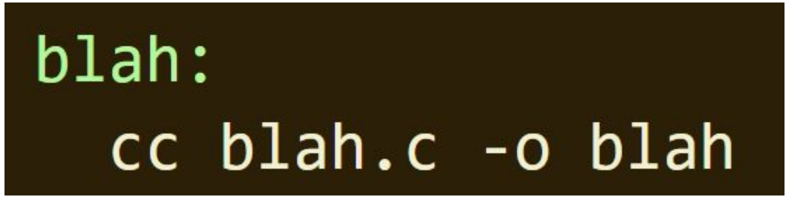
Typical Makefile

Let's create typical Makefile - one that compiles a single C file. But before we do, make a file called blah.c that has the following contents:



```c
// blah.c
int main() { return 0; }
```

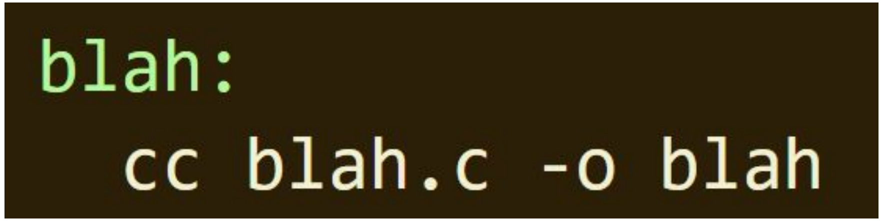
Then create the Makefile (called Makefile, as always):



```makefile
blah:
  cc blah.c -o blah
```

Running this Makefile

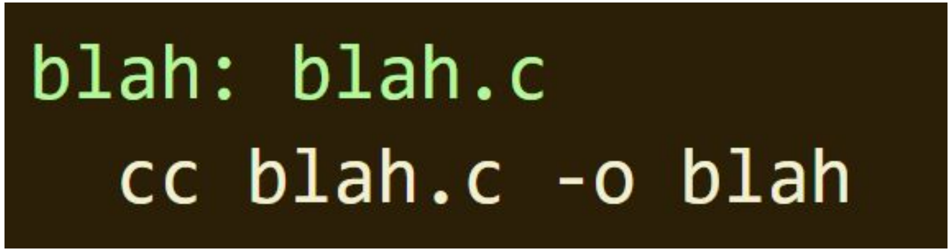
- To run this we would simply run make.
- Since there's no target supplied as an argument to the make command, the first target is run.
    - In this case, there's only one target (blah).
    - The first time you run this, blah will be created.
- The second time we run make, you'll see
    - make: 'blah' is up to date.
- That's because the blah file already exists. But there's a problem: if we modify blah.c and then run make, nothing gets recompiled.



```
blah:
	cc blah.c -o blah
```

Fixing this Makefile

We solve this by adding a prerequisite:



```
blah: blah.c
	cc blah.c -o blah
```

When we run make again, the following set of steps happens:

- The first target is selected, because the first target is the default target
- This has a prerequisite of blah.c
- Make decides if it should run the blah target. It will only run if blah doesn't exist, or blah.c is newer than blah
- This last step is critical, and is the essence of make. **What it's attempting to do is decide if the prerequisites of blah have changed since blah was last compiled.** That is, if blah.c is modified, running make should recompile the file. And conversely, if blah.c has not changed, then it should not be recompiled.

How do Makefiles determine when to / not to compile?

To make this happen, it uses the filesystem timestamps as a proxy to determine if something has changed. This is a reasonable heuristic, because file timestamps typically will only change if the files are modified. But it's important to realize that this isn't always the case. You could, for example, modify a file, and then change the modified timestamp of that file to something old. If you did, Make would incorrectly guess that the file hadn't changed and thus could be ignored.

A more complex makefile example

The following Makefile ultimately runs all three targets. When you run make in the terminal, it will build a program called blah in a series of steps:

- Make selects the target blah, because the first target is the default target
- blah requires blah.o, so make searches for the blah.o target
- blah.o requires blah.c, so make searches for the blah.c target

```
blah: blah.o
  cc blah.o -o blah # Runs third

blah.o: blah.c
  cc -c blah.c -o blah.o # Runs second

# Typically blah.c would already exist, but I want to limit any additional required files
blah.c:
  echo "int main() { return 0; }" > blah.c # Runs first
```

A more complex makefile example

- blah.c has no dependencies, so the echo command is run
- The cc -c command is then run, because all of the blah.o dependencies are finished
- The top cc command is run, because all the blah dependencies are finished
- That's it: blah is a compiled c program

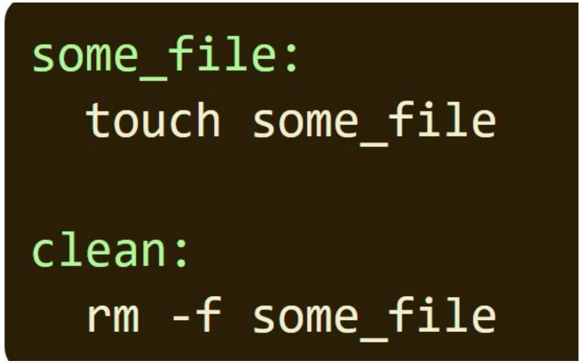
```
blah: blah.o
  cc blah.o -o blah # Runs third

blah.o: blah.c
  cc -c blah.c -o blah.o # Runs second

# Typically blah.c would already exist, but I want to limit any additional required files
blah.c:
  echo "int main() { return 0; }" > blah.c # Runs first
```

Make Clean

clean is often used as a target that removes the output of other targets, but it is not a special word in Make. You can run make and make clean on this to create and delete some_file.

Note that clean is doing two new things here:

- It's a target that is not first (the default), and not a prerequisite. That means it'll never run unless you explicitly call make clean
- It's not intended to be a filename. If you happen to have a file named clean, this target won't run, which is not what we want.

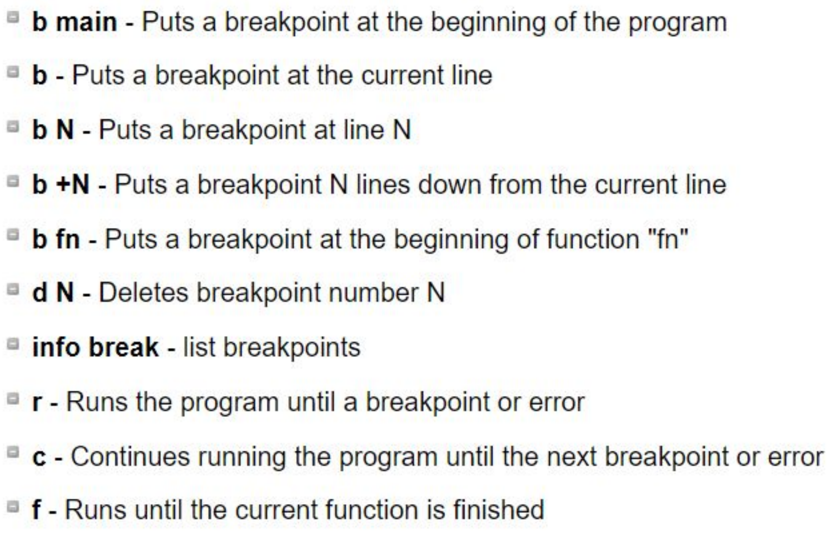


```
some_file:
  touch some_file

clean:
  rm -f some_file
```

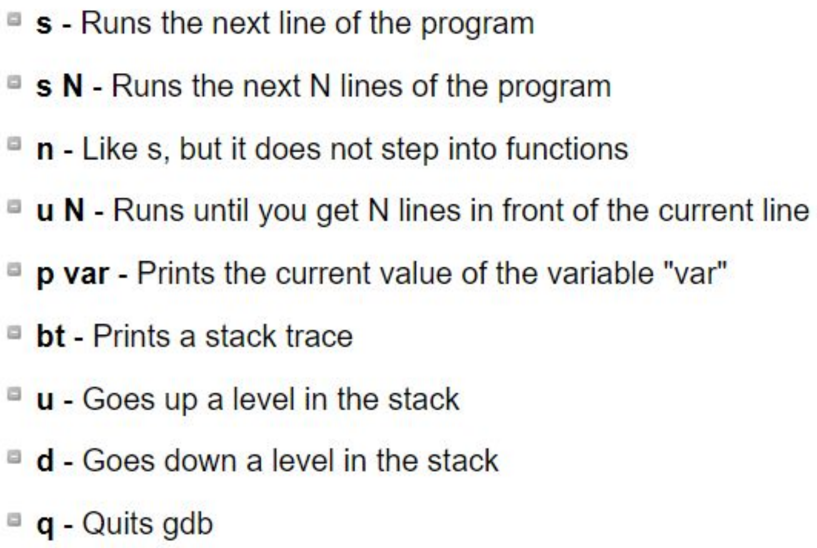
Debugging Tools

GDB Cheat Sheet

1. Run a program with gdb: `gdb <executable>` e.g. `gdb ./test_suite`
2. Start the program in gdb: `start <arg1> <arg2> ..` e.g. `start 4`



- **b main** - Puts a breakpoint at the beginning of the program
- **b** - Puts a breakpoint at the current line
- **b N** - Puts a breakpoint at line N
- **b +N** - Puts a breakpoint N lines down from the current line
- **b fn** - Puts a breakpoint at the beginning of function "fn"
- **d N** - Deletes breakpoint number N
- **info break** - list breakpoints
- **r** - Runs the program until a breakpoint or error
- **c** - Continues running the program until the next breakpoint or error
- **f** - Runs until the current function is finished



- **s** - Runs the next line of the program
- **s N** - Runs the next N lines of the program
- **n** - Like s, but it does not step into functions
- **u N** - Runs until you get N lines in front of the current line
- **p var** - Prints the current value of the variable "var"
- **bt** - Prints a stack trace
- **u** - Goes up a level in the stack
- **d** - Goes down a level in the stack
- **q** - Quits gdb

GDB Cheat Sheet: breakpoint example

1. `b Deque.c:122` → set a breakpoint on line 122 in Deque.c
2. `c` -> continue the program until the breakpoint
3. `s` -> step through each line at a time
4. Once the breakpoint is hit, print variables accordingly with `p`
5. In case of segfaults, type `where` to view where it is

Valgrind Cheat Sheet

For a program prog that can take flag a

Default syntax: valgrind ./prog -a

Flags:

- --leak-check=full
- --show-leak-kinds=all
- --track-origins=yes
- --verbose

That’s all we have for today!

Reminders:

- TA-lead recitations will take place on
  - Tuesdays 6:30-8:00pm in Moore 100A
  - Wednesday 12:00-1:30pm in Moore 100C
- HW7 is due this Friday 11/11 at 11:59pm