
NATIONAL RESEARCH UNIVERSITY

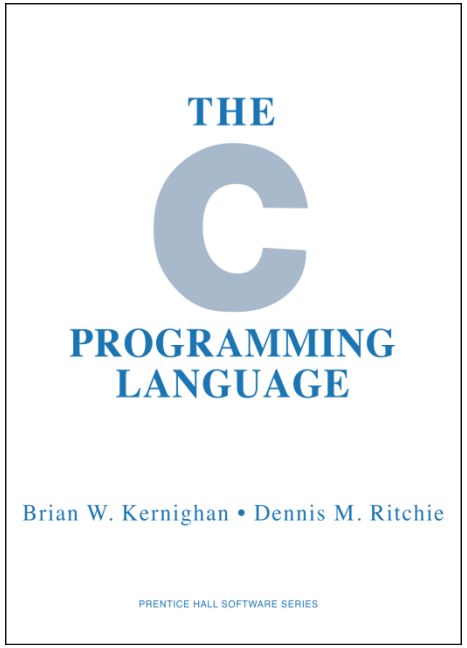

# Computer Architecture and Operating Systems
# Lecture 2: The C Programming Language

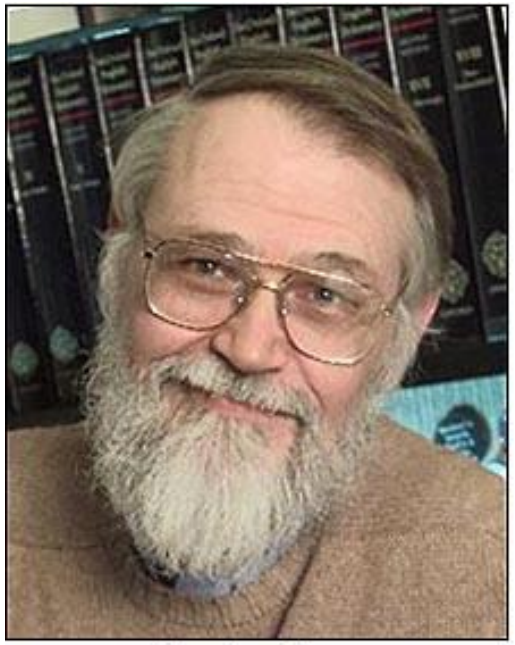
**Andrei Tatarnikov**

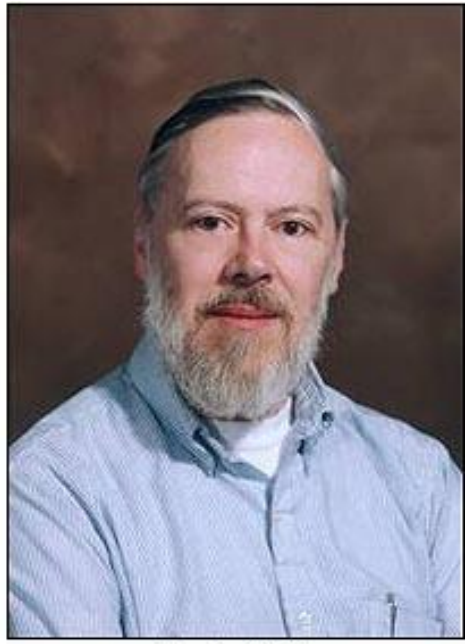
atatarnikov@hse.ru
@andrewt0301

# The C Programming Language

- 1972-1973: Developed at Bell Labs by Dennis Ritchie to create utilities for Unix
- 1973: Unix was re-implemented in C
- 1978: Brian Kernighan and Dennis Ritchie published The C Programming Language
- 1989/1990: ANSI C and ISO C; 1999: C99; 2011: C11; 2017: C17





Brian Kernighan


Dennis Ritchie

# The Application of C Language

- C is not a “very high level” language, nor a “big” one, and is not specialized to any particular area of application. But its absence of restrictions and its generality make it more convenient and effective for many tasks than supposedly more powerful languages.

**Kernighan and Ritchie**

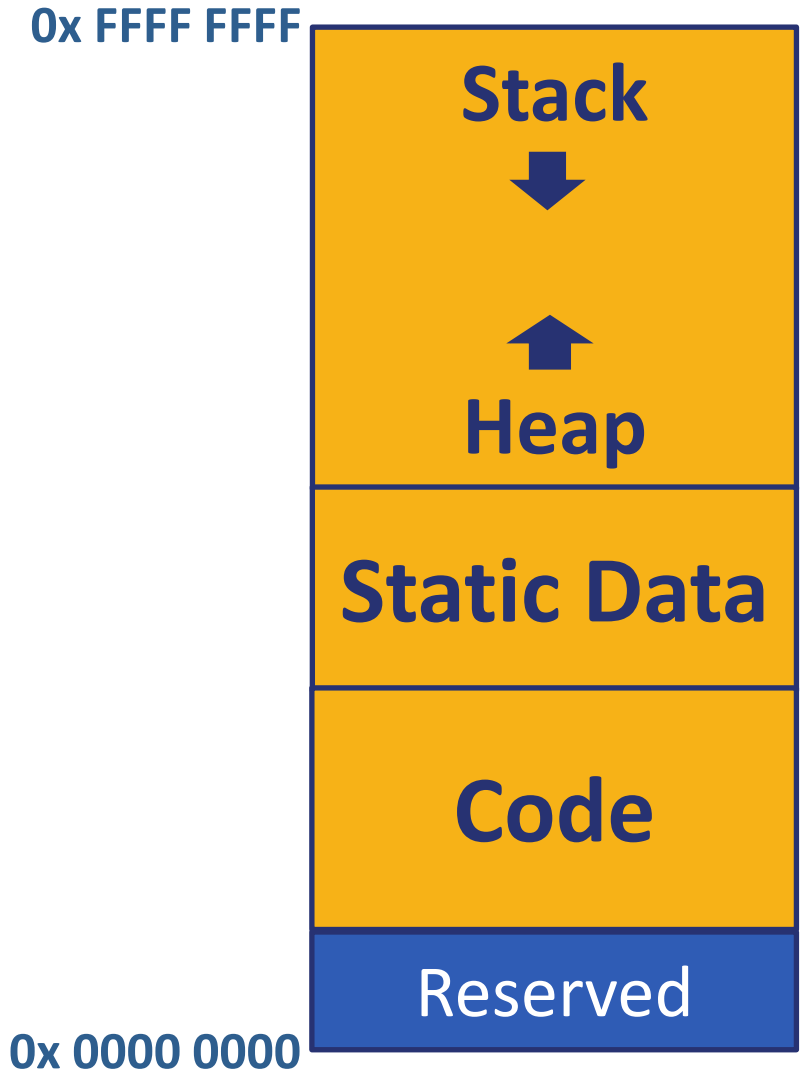
- With C we can write programs that allow us to exploit underlying features of the architecture

# C Concepts

| | |
|---|---|
| **Compiler** | Creates usable programs from C source code |
| **Typed variables** | Must declare the kind of data the variable will contain |
| **Typed functions** | Must declare the kind of data returned from the function |
| **Header files (.h)** | Allows declaring functions and variables in separate files |
| **Structs** | Groups of related values |
| **Enums** | Lists of predefined values |
| **Pointers** | Aliases to other variables |

# C Memory Layout

- Program's **address space** contains 4 regions:
  - **Stack**: local variables, grows downward
  - **Heap**: space requested via *malloc*() and used with pointers; resizes dynamically, grows upward
  - **Static Data**: global and static variables, does not grow or shrink
  - **Code**: loaded when program starts, does not change




**OS prevents accesses between stack and heap (via virtual memory)**

# Where Do the Variables Go?

- Declared outside a function:
  - **Static Data**
- Declared inside a function:
  - **Stack**
    - main() is a function
    - freed when the function returns
- Dynamically allocated:
  - **Heap**
    - i.e. malloc (will be covered shortly)

```
#include <stdio.h>

int varGlobal;

int main() {
  int varLocal;
  int *varDyn =
    malloc(sizeof(int));
}
```

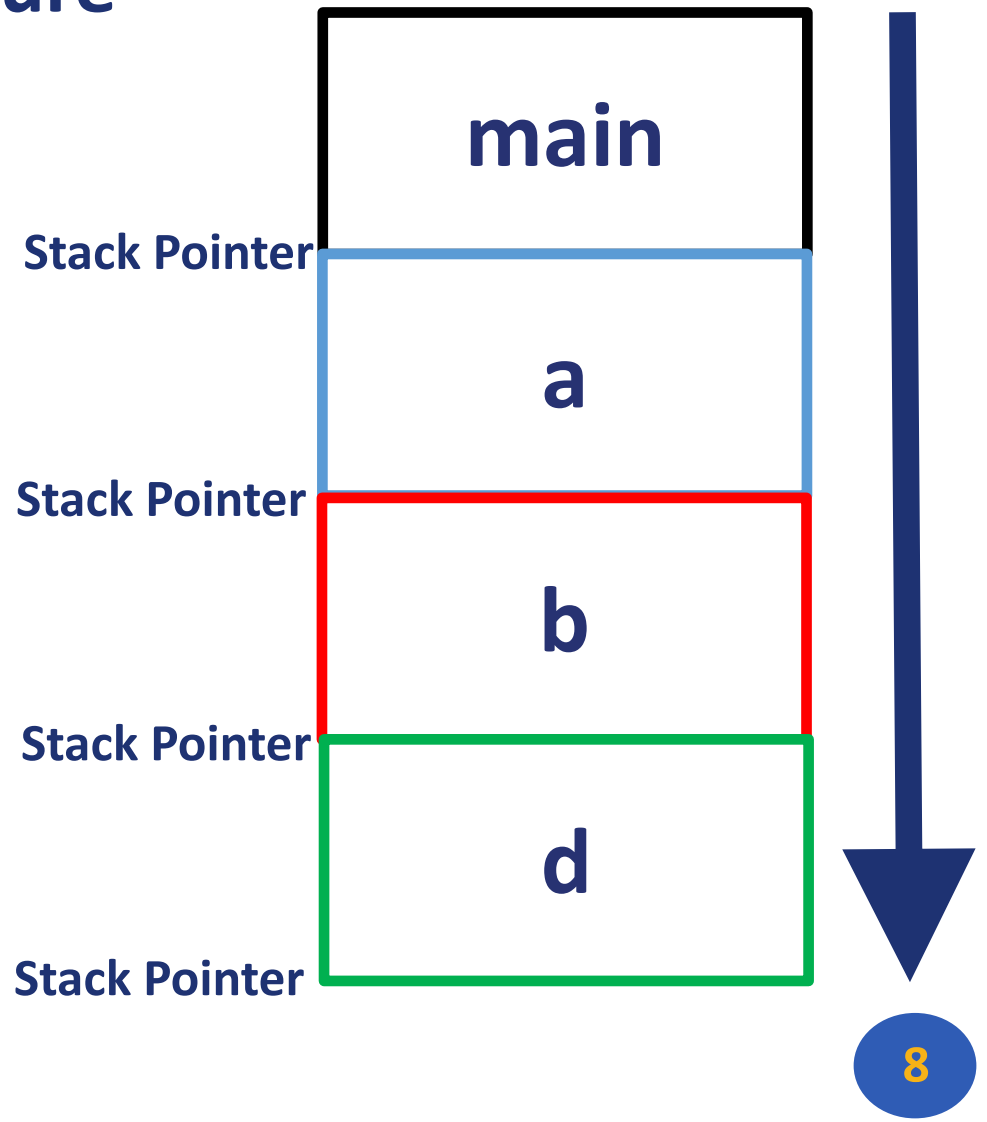
# Stack

- Each stack frame is a contiguous block of memory holding the local variables of a single function
- A stack frame includes:
  - Location of caller function
  - Function arguments
  - Space for local variables
- Stack pointer (SP) tells where lowest (current) stack frame is
- When function ends, stack pointer is moved back (but data remains (garbage!)); frees memory for future stack frames

# Stack

## Last In, First Out (LIFO) data structure

```
int main() {
    a(0);
    return 1;
}
void a(int m) {
    b(1);
}
void b(int n) {
    c(2);
    d(4);
}
void c(int o) {
    printf("c");
}
void d(int p) {
    printf("d");
}
```



# Stack Misuse

```
int *getPtr() {
    int y;
    y = 3;
    return &y;
}

int main () {
    int *stackAddr, content;
    stackAddr = getPtr();
    content = *stackAddr;
    printf("%d", content); /* 3 */
    content = *stackAddr;
    printf("%d", content); /* ? */
}
```

**Never return pointers to local variable from functions!**

**Your compiler will warn you about this.**

**Do not ignore such warnings!**

**printf overwrites stack frames.**

# Static Data

- Place for variables that persist
  - Data not subject to comings and goings like function calls
  - Examples: string literals, global variables
  - String literal example: char * str = “hi”;
  - Do not be mistaken with: char str[] = “hi”;
    - This will put str on the stack!
- Size does not change, but sometimes data can
  - Notably string literals cannot

# Code

- Copy of your code goes there
  - C code becomes data too!
- Does (should) not change
  - Typically read-only

# Dynamic Memory Allocation

- Want persisting memory (like static) even when we do not know size at compile time?
  - e.g.  input files, user interaction
  - Stack will not work because stack frames are not persistent
- Dynamically allocated memory goes on the Heap
  - more permanent than Stack
- Need as much space as possible without interfering with Stack
  - Start at opposite end and grow towards Stack

# The sizeof Operator

- If integer sizes are machine dependent, how do we tell?
- Use sizeof() operator
  - Returns size in number of char-sized units of a variable or data type name
    - Examples: **int x; sizeof(x); sizeof(int);**
  - sizeof(char) is always 1
- Can we use sizeof to determine a length of an array?
  - Generally **no** but there is an exception:
    - **int a[61];**
    - **sizeof(a)** gets the total number of bytes stored in the array a.
    - To get the number of elements, use: **sizeof(a) / sizeof(int)**
    - This **ONLY** works for arrays defined on the stack **IN THE SAME FUNCTION**
  - It is not recommended to do this. A preferred way is to keep track of an array size elsewhere.

# Allocating Memory

- Functions for requesting memory: malloc(), calloc(), and realloc()
- malloc(n)
  - Allocates a continuous block of n bytes of uninitialized memory (contains garbage!)
  - Returns a pointer to the beginning of the allocated block; NULL indicates failed request (check for this!)
  - Different blocks not necessarily adjacent

# Any Questions?

```
            .text
__start:    addi t1, zero, 0x18
            addi t2, zero, 0x21
cycle:      beq t1, t2, done
            slt t0, t1, t2
            bne t0, zero, if_less
            nop
            sub t1, t1, t2
            j cycle
            nop
if_less:    sub t2, t2, t1
            j cycle
done:       add t3, t1, zero
```