Please check your attendance using Blackboard!

Lecture 3 – Memory Safety

[COSE451] Software Security

Instructor: Seunghoon Woo

Spring 2024

Overview

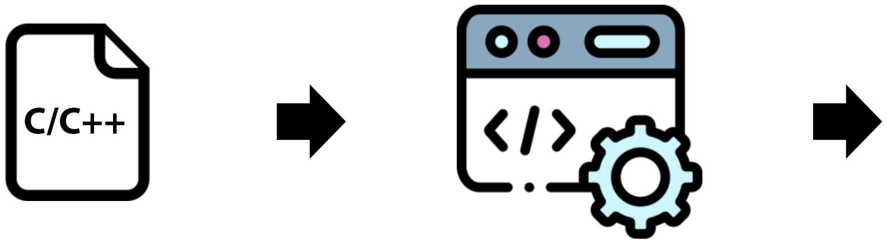
- **Memory safety**

Memory safety

- **Memory?**
  - The space in a computer where programs or data can be stored and accessed
- **Memory safety?**
  - Ensuring the integrity of a program's data structures
    - Preventing attackers from reading or writing to memory locations other than those intended by the programmer
  - Preventing problems that arise owing to improper memory management

Memory safety

- **Memory structure**



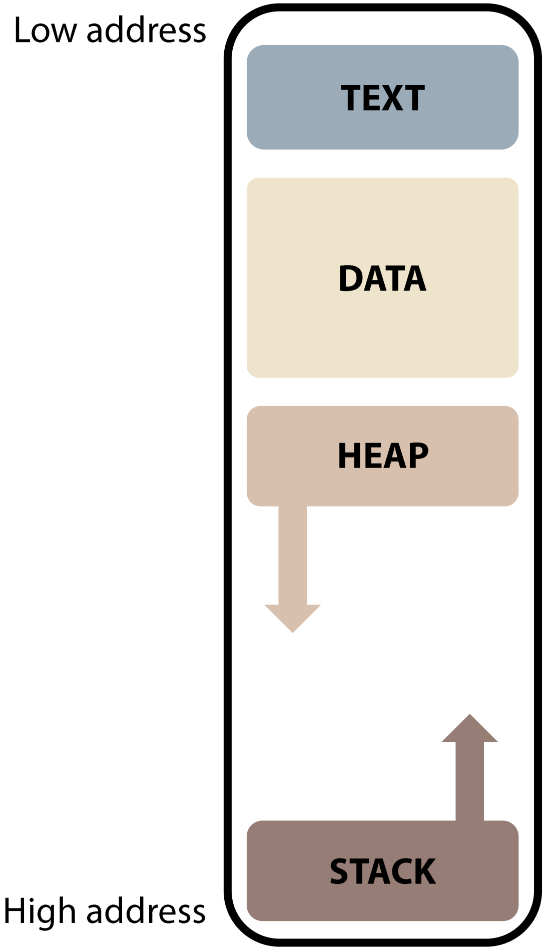
Execute!

1. Loaded into memory
2. Writes and reads data into/from memory

   based on the content written in the code

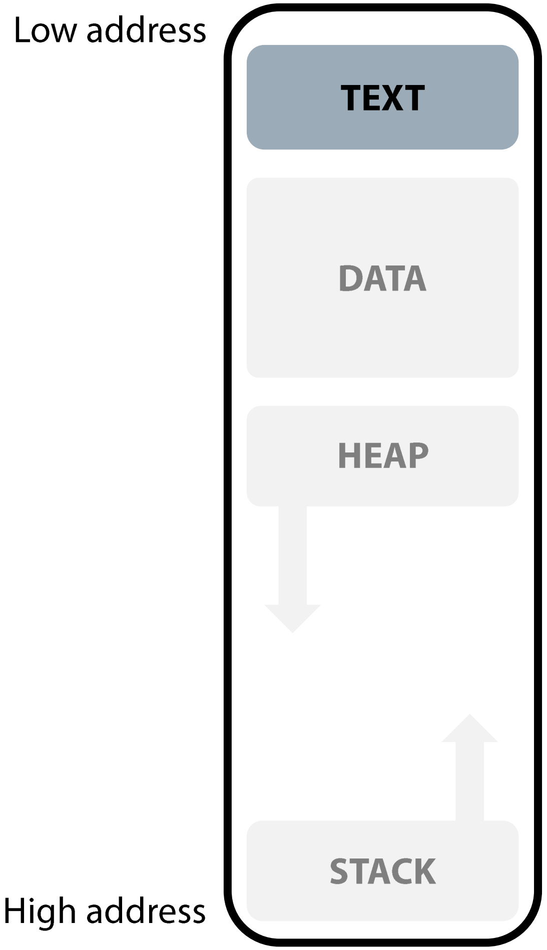
Memory safety

- **Memory structure**
  - The typical memory space allocated to a program by the operating system



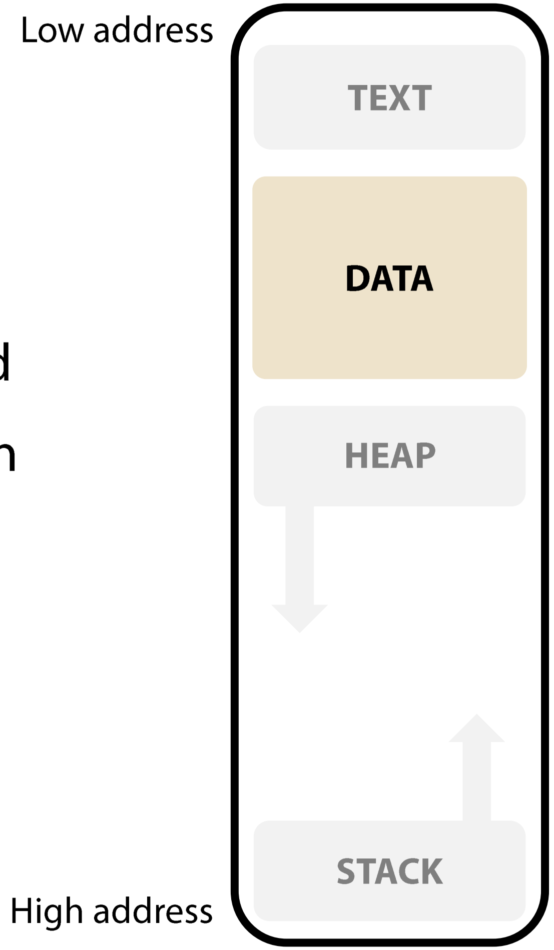
Memory safety

- **Memory structure**
  - TEXT (CODE)
    - The area where the executable code is stored
    - CPU fetches and processes instructions stored in this section one by one
    - E.g., conditional statements, functions, constants, …



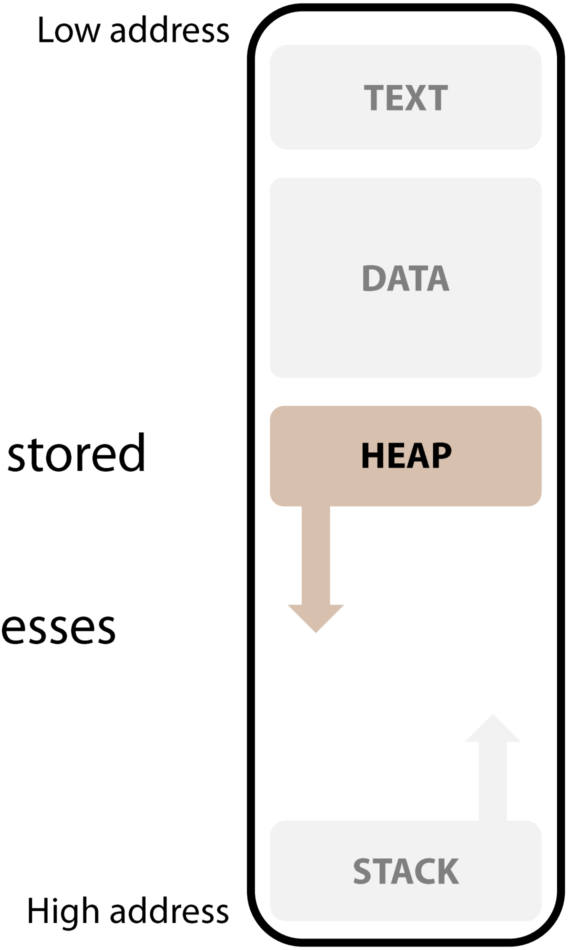
Memory safety

- **Memory structure**
  - DATA
    - The area where global and static variables are stored
    - Variables typically declared before the main function (prior to program execution) that persist in memory until the program ends



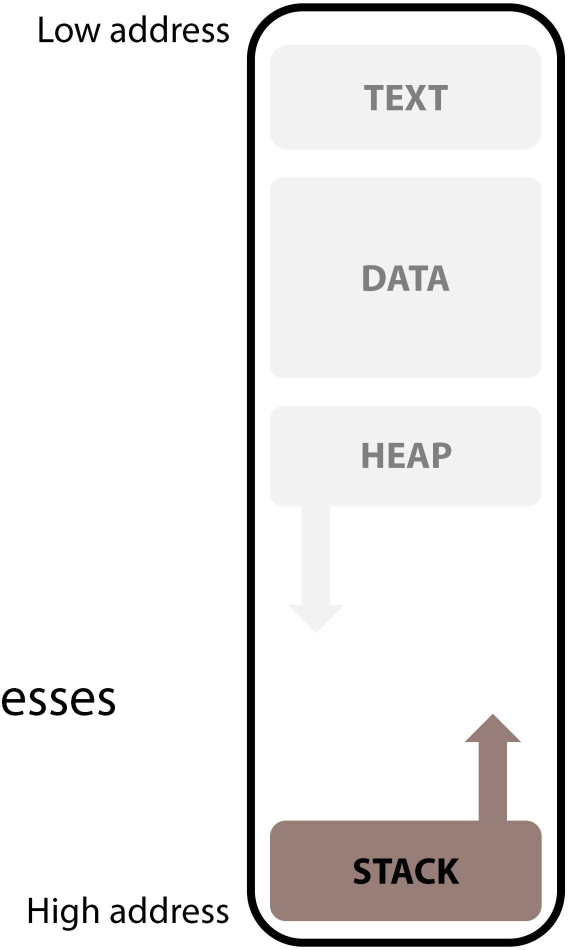
Memory safety

- **Memory structure**
  - HEAP
    - User-managed memory area
    - Location where dynamically allocated variables are stored
      - E.g., malloc
    - Allocated (loaded) from low addresses to high addresses



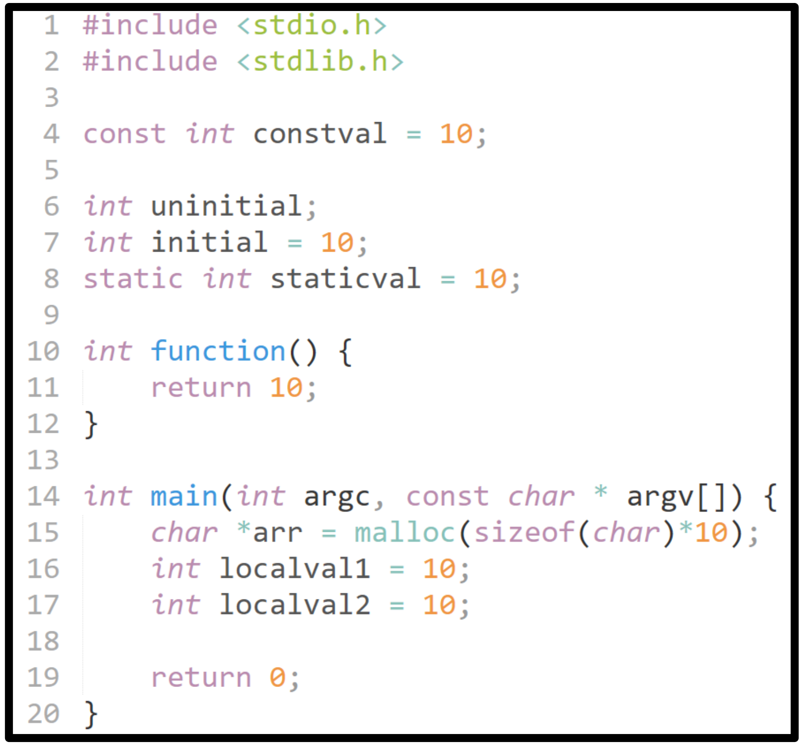
Memory safety

- **Memory structure**
  - STACK
    - The area where local variables and parameters associated with function calls are stored
    - Allocated during a function call and deallocated (destroyed) when the function call is complete
    - Allocated (loaded) from high addresses to low addresses

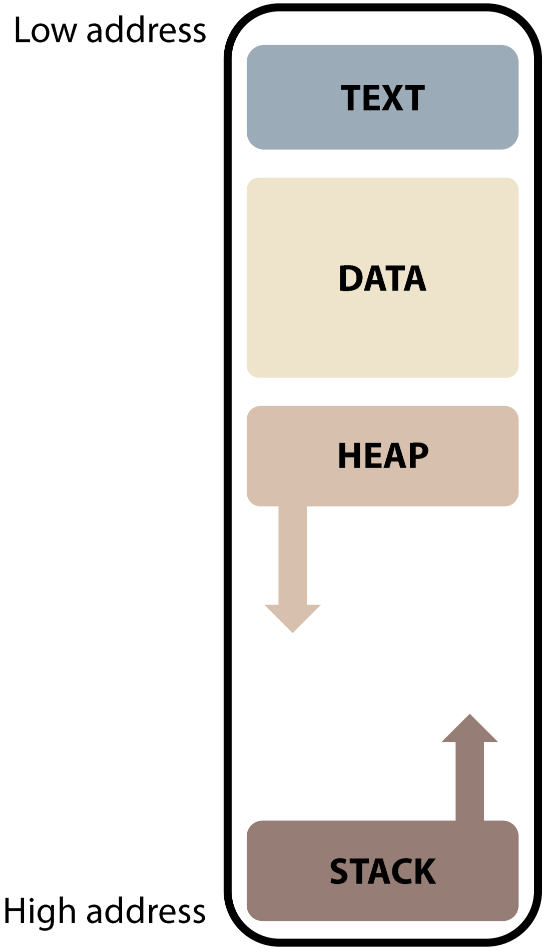


Memory safety

- **Memory structure**
  - Example

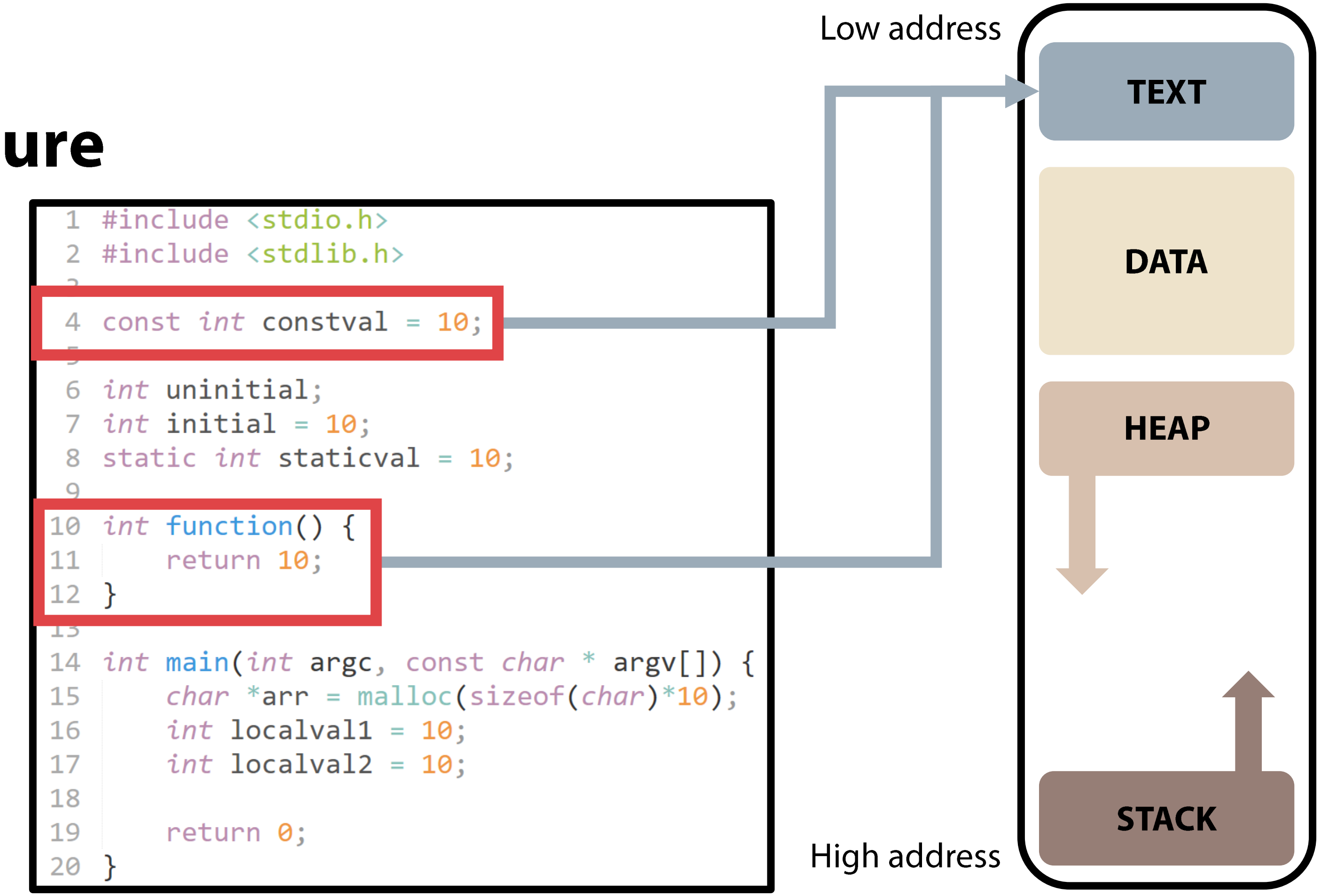


```c
#include <stdio.h>
#include <stdlib.h>

const int constval = 10;

int uninitial;
int initial = 10;
static int staticval = 10;

int function() {
    return 10;
}

int main(int argc, const char * argv[]) {
    char *arr = malloc(sizeof(char)*10);
    int localval1 = 10;
    int localval2 = 10;

    return 0;
}
```



Memory safety

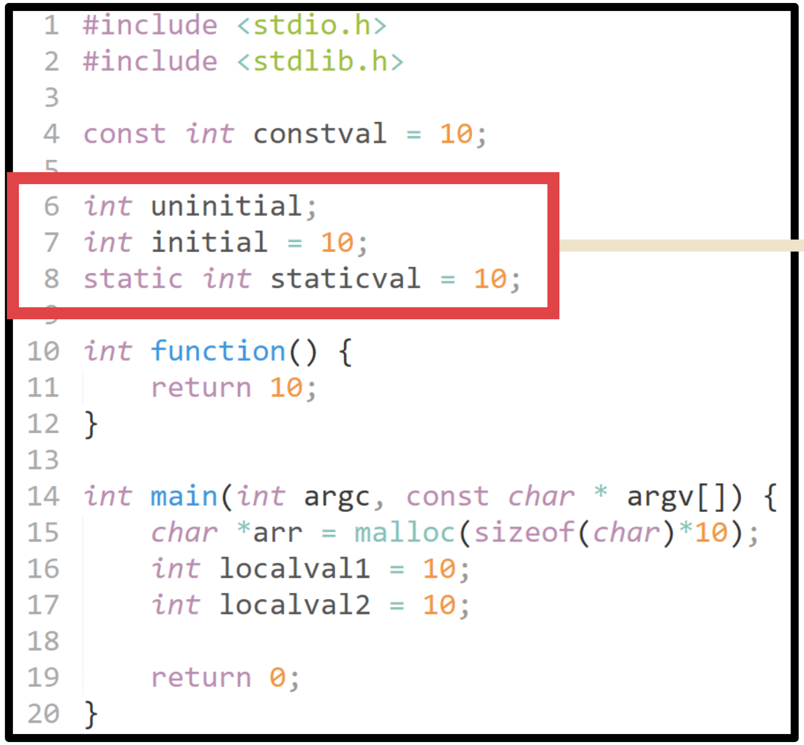
- **Memory structure**
  - Example



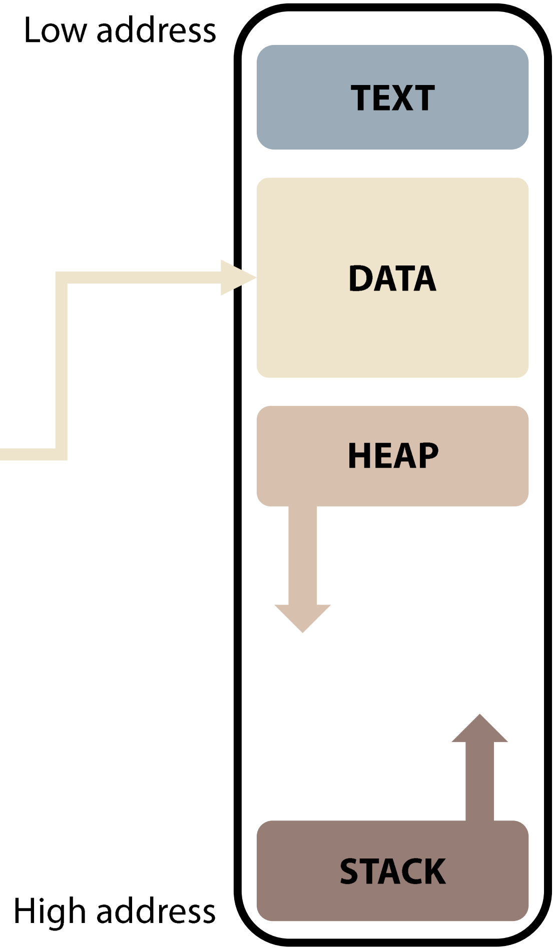
```c
#include <stdio.h>
#include <stdlib.h>

const int constval = 10;

int uninitial;
int initial = 10;
static int staticval = 10;

int function() {
    return 10;
}

int main(int argc, const char * argv[]) {
    char *arr = malloc(sizeof(char)*10);
    int localval1 = 10;
    int localval2 = 10;

    return 0;
}
```

Memory safety

- **Memory structure**
  - Example

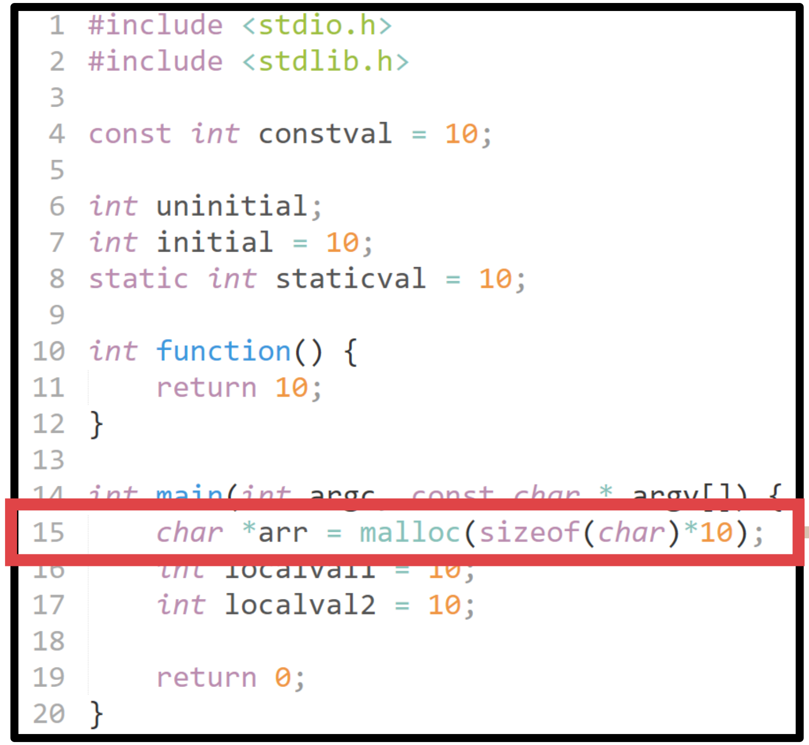


```c
#include <stdio.h>
#include <stdlib.h>

const int constval = 10;

int uninitial;
int initial = 10;
static int staticval = 10;

int function() {
    return 10;
}

int main(int argc, const char * argv[]) {
    char *arr = malloc(sizeof(char)*10);
    int localval1 = 10;
    int localval2 = 10;

    return 0;
}
```

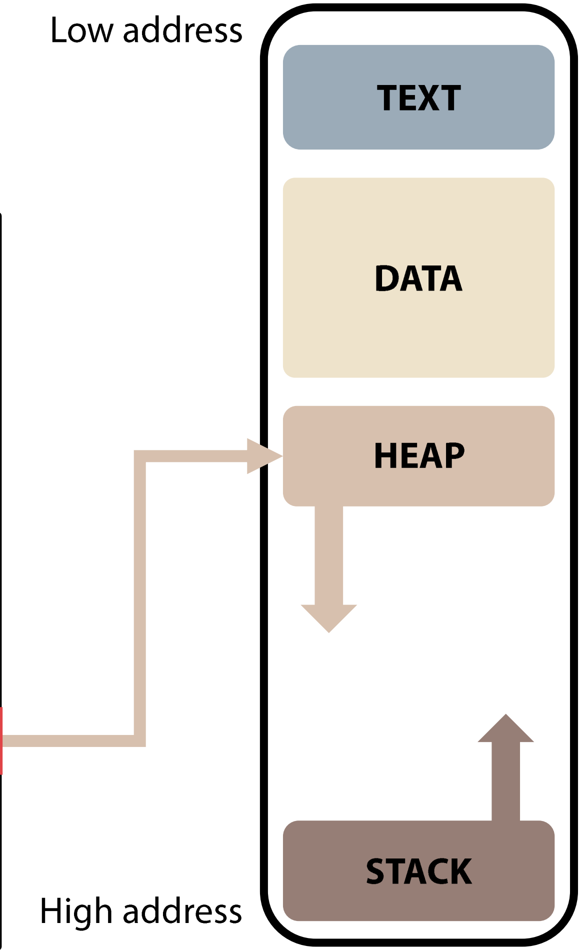


Memory safety

- **Memory structure**
  - Example

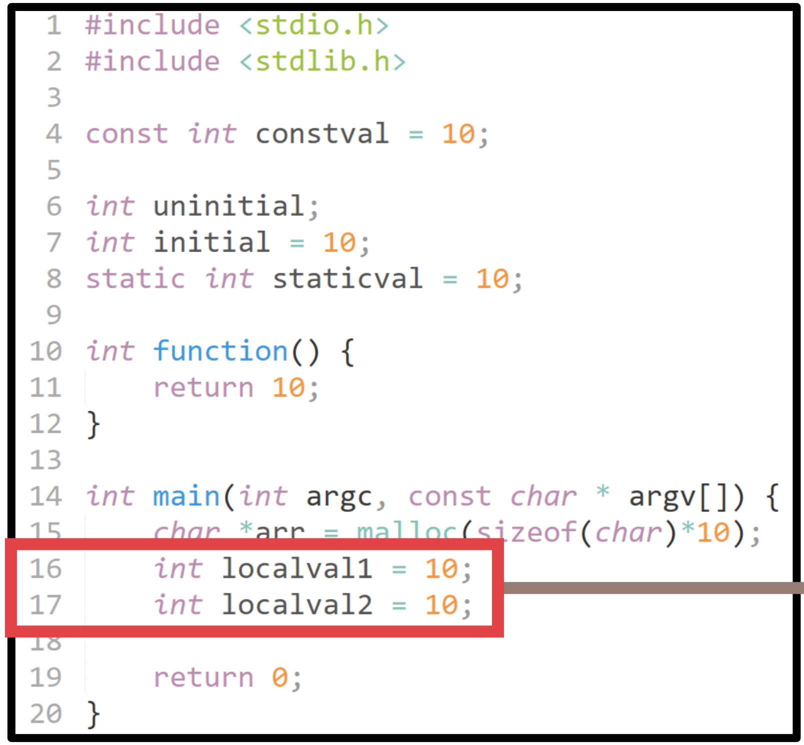


```c
#include <stdio.h>
#include <stdlib.h>

const int constval = 10;

int uninitial;
int initial = 10;
static int staticval = 10;

int function() {
    return 10;
}

int main(int argc, const char * argv[]) {
    char *arr = malloc(sizeof(char)*10);
    int localval1 = 10;
    int localval2 = 10;

    return 0;
}
```

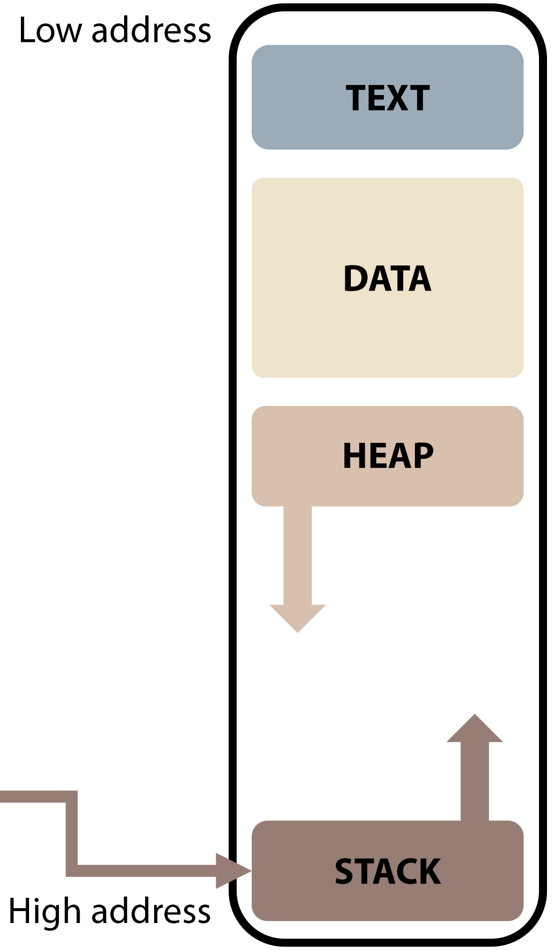


Memory safety

- **Memory structure**
  - Example



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

const int constval = 10;

int uninitial;
int initial = 10;
static int staticval = 10;

int function() {
    return 10;
}

int main(int argc, const char * argv[]) {
    char *arr = malloc(sizeof(char)*10);
    int localval1 = 10;
    int localval2 = 10;

    return 0;
}
```



Memory safety

- **Memory structure**
  - Example: memory addresses

Constant memory address	`0x56557008`	TEXT
Uninitialized variable memory address	`0x56559014`	DATA
Initialized variable memory address	`0x56559008`	DATA
Static variable memory address	`0x5655900c`	DATA
Function memory address	`0x5655619d`	TEXT
Dynamically-allocated variable memory address	`0x5655a1a0`	HEAP
Local variable 1 memory address	`0xffffd0b8`	STACK
Local variable 2 memory address	`0xffffd0b4`	STACK

Memory safety

- **Buffer overflow**
  - A buffer refers to a temporary storage space
  - Inputting data larger than a certain size into a buffer of a fixed size
  - Overflowing the buffer can lead to the followings
    (1) Corruption of the memory area
    (2) Potential for stealing hidden information
    (3) An attacker can execute the desired code

Memory safety

- **Buffer overflow**
  - Focusing on two types of buffer overflow
    - Stack buffer overflow
    - Heap buffer overflow

Memory safety

- **Stack buffer overflow**
  - ▪ Security issue that occurs when the memory in the stack area exceeds the specified range
    - E.g., inserting a value larger than the allocated variable size

※ Stack overflow

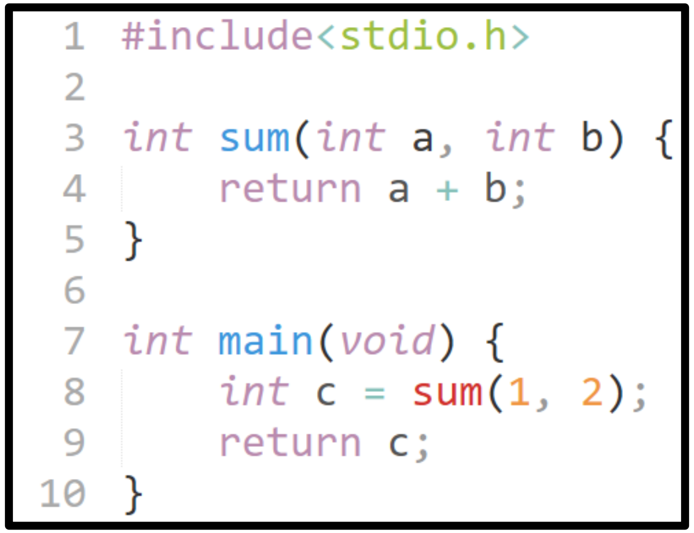
- ▪ A bug caused by excessive expansion of the stack area
  - E.g., infinite recursive function calls

Memory safety

- **Stack frame**
  - The space created to distinguish the stack area specific to each function when the function is called
    - This stores local variables and parameters related to the function
    - This is allocated during a function call, and is deallocated when the function ends

Memory safety

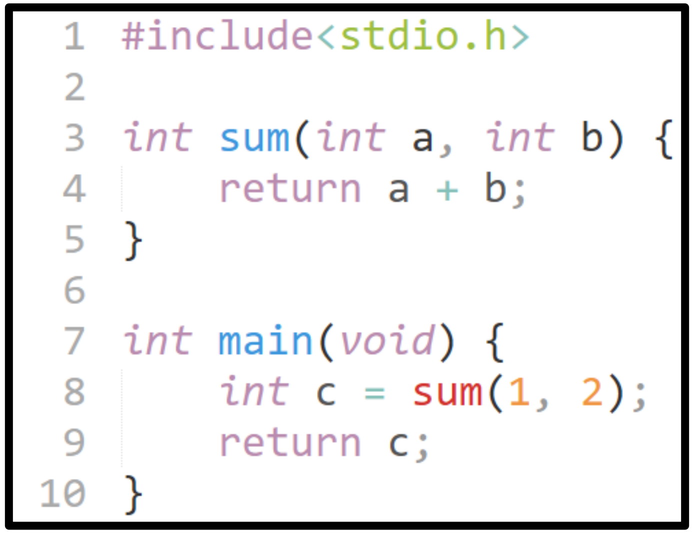
- **Stack frame**
  - Example



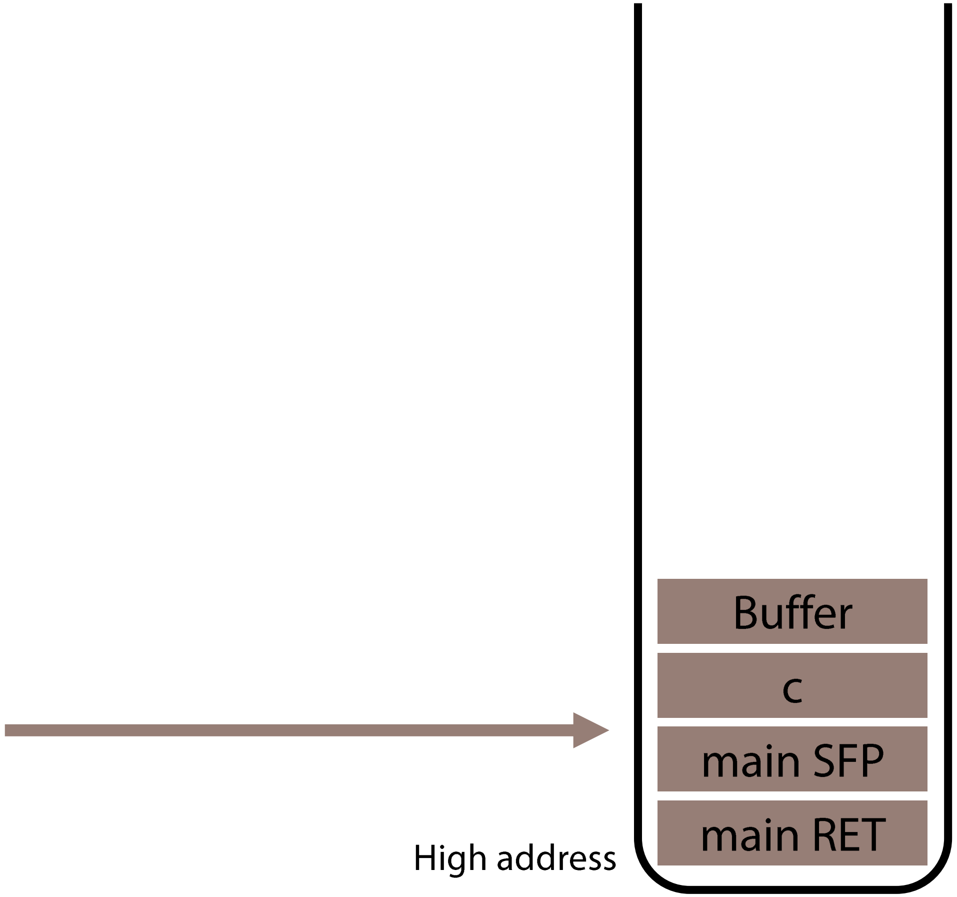
```c
#include<stdio.h>

int sum(int a, int b) {
    return a + b;
}

int main(void) {
    int c = sum(1, 2);
    return c;
}
```

Memory safety

- **Stack frame**
  - Example



```c
#include<stdio.h>

int sum(int a, int b) {
    return a + b;
}

int main(void) {
    int c = sum(1, 2);
    return c;
}
```



Buffer

c

main SFP

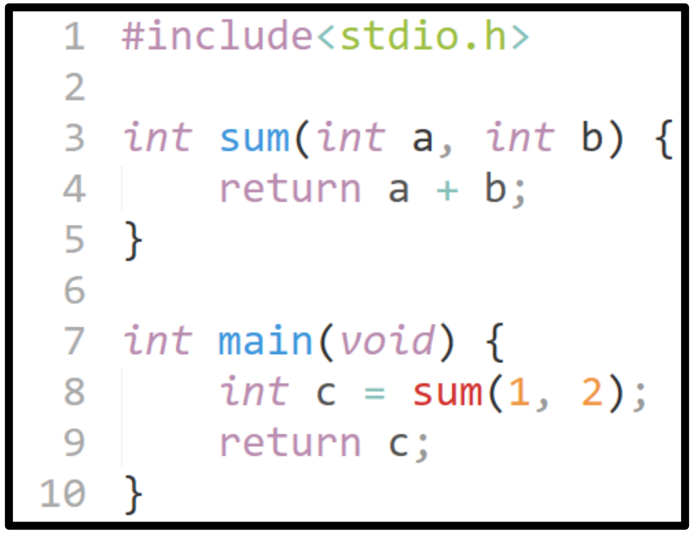
main RET

High address

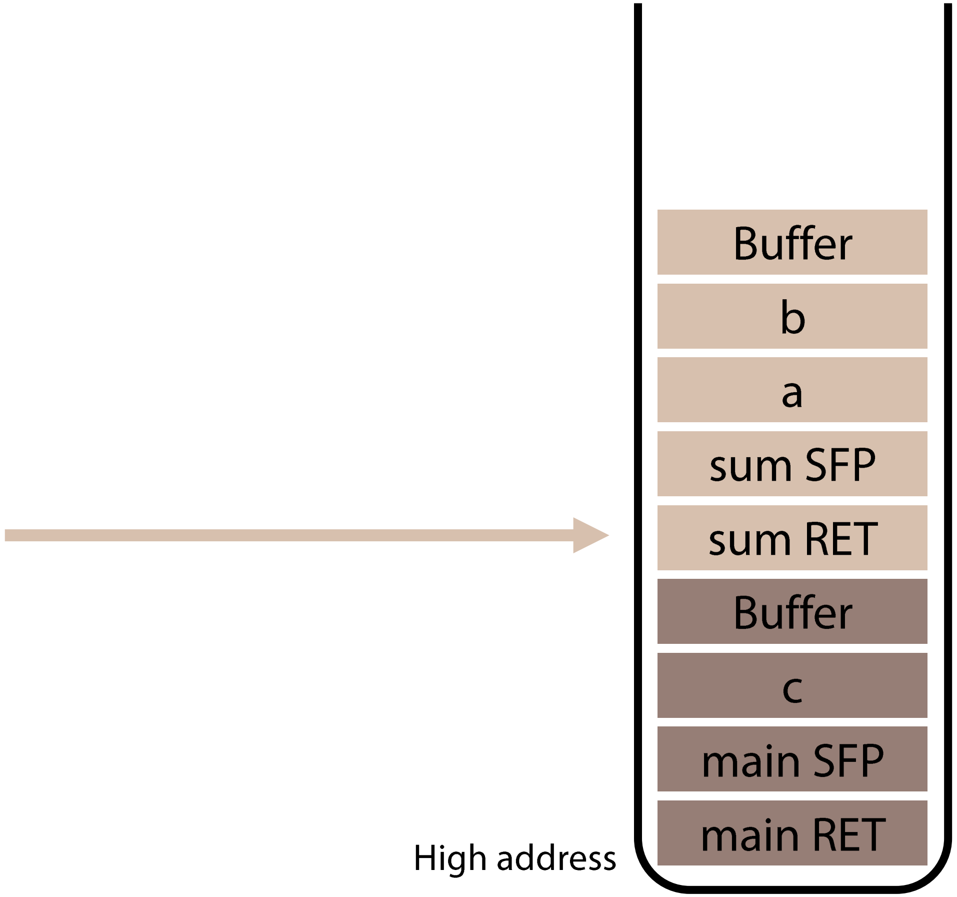
*SFP: Stack Frame Pointer

Memory safety

- **Stack frame**
  - Example

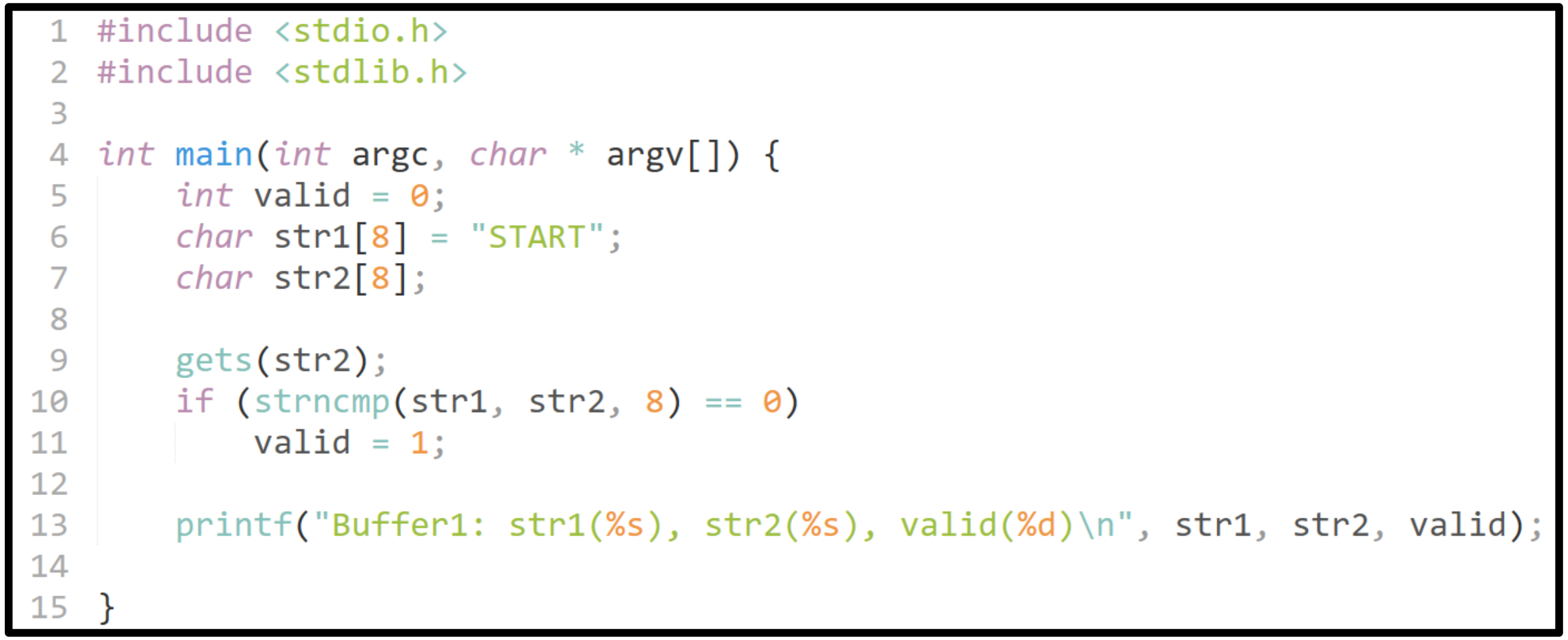


```c
#include<stdio.h>

int sum(int a, int b) {
    return a + b;
}

int main(void) {
    int c = sum(1, 2);
    return c;
}
```



Memory safety

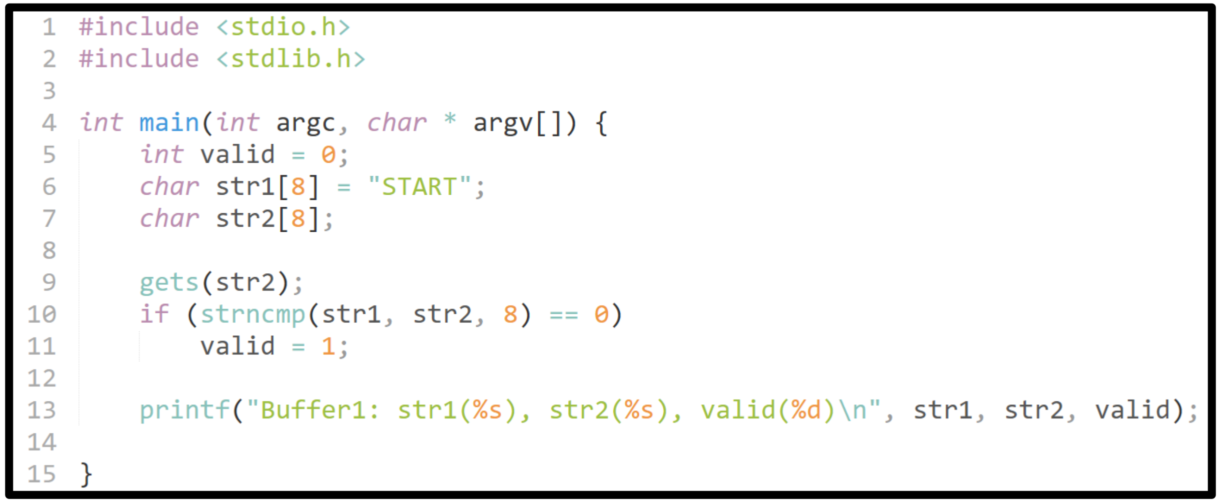
- **Stack buffer overflow**
  - Example



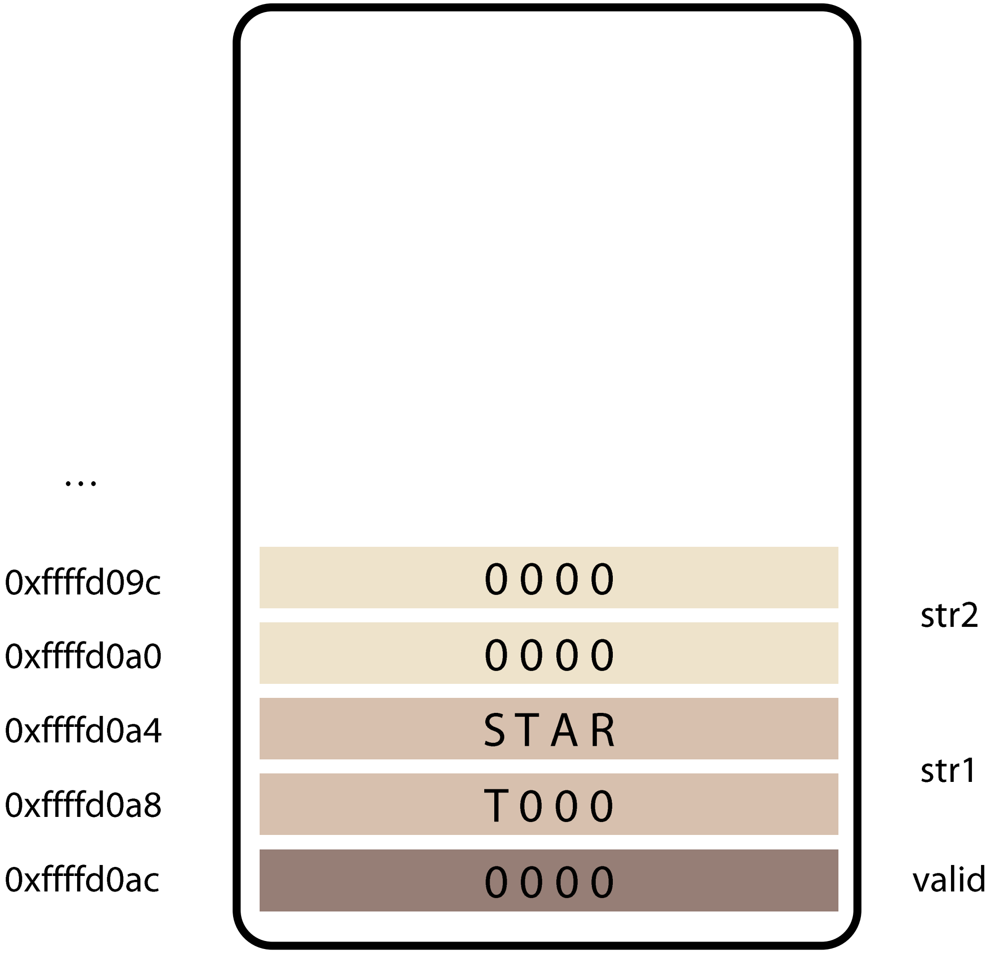
```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char * argv[]) {
    int valid = 0;
    char str1[8] = "START";
    char str2[8];

    gets(str2);
    if (strncmp(str1, str2, 8) == 0)
        valid = 1;

    printf("Buffer1: str1(%s), str2(%s), valid(%d)\n", str1, str2, valid);

}
```

Memory safety

- **Stack buffer overflow**
  - Example



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

int main(int argc, char * argv[]) {
    int valid = 0;
    char str1[8] = "START";
    char str2[8];

    gets(str2);
    if (strncmp(str1, str2, 8) == 0)
        valid = 1;

    printf("Buffer1: str1(%s), str2(%s), valid(%d)\n", str1, str2, valid);

}
```

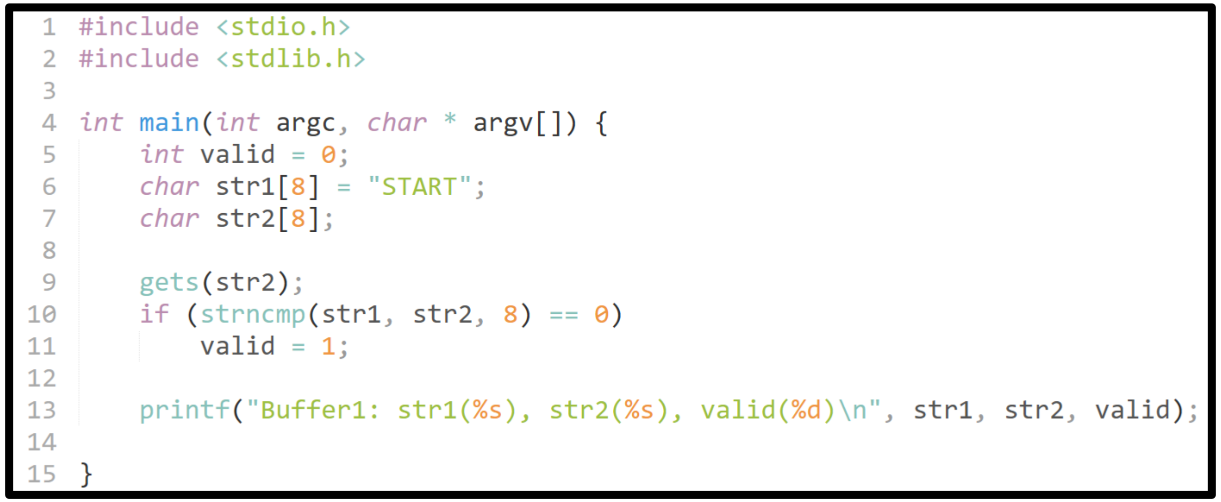


...

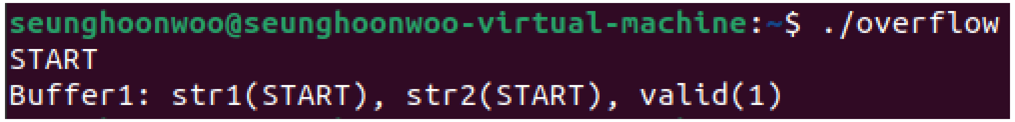
0xffffd09c	0 0 0 0	str2
0xffffd0a0	0 0 0 0	
0xffffd0a4	S T A R	str1
0xffffd0a8	T 0 0 0	
0xffffd0ac	0 0 0 0	valid

Memory safety

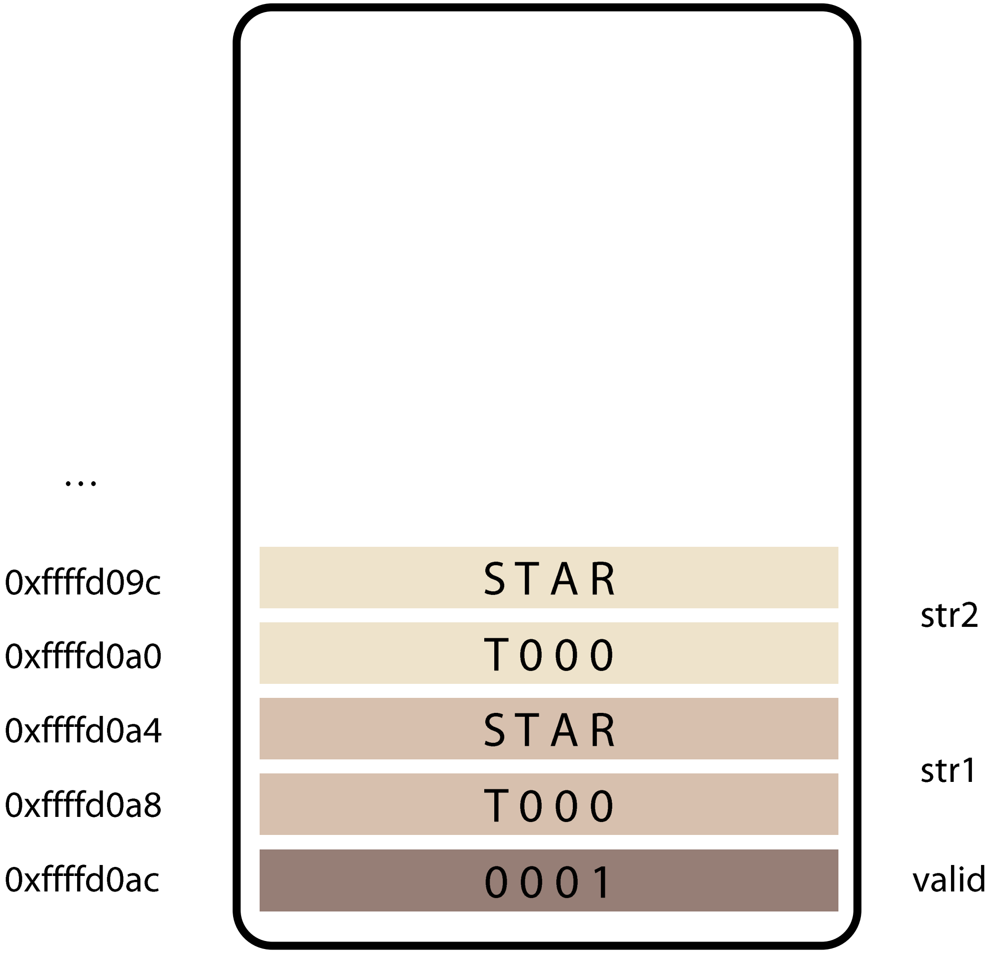
- **Stack buffer overflow**
  - Example: input = “START” (no problem)



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

int main(int argc, char * argv[]) {
    int valid = 0;
    char str1[8] = "START";
    char str2[8];

    gets(str2);
    if (strncmp(str1, str2, 8) == 0)
        valid = 1;

    printf("Buffer1: str1(%s), str2(%s), valid(%d)\n", str1, str2, valid);

}
```



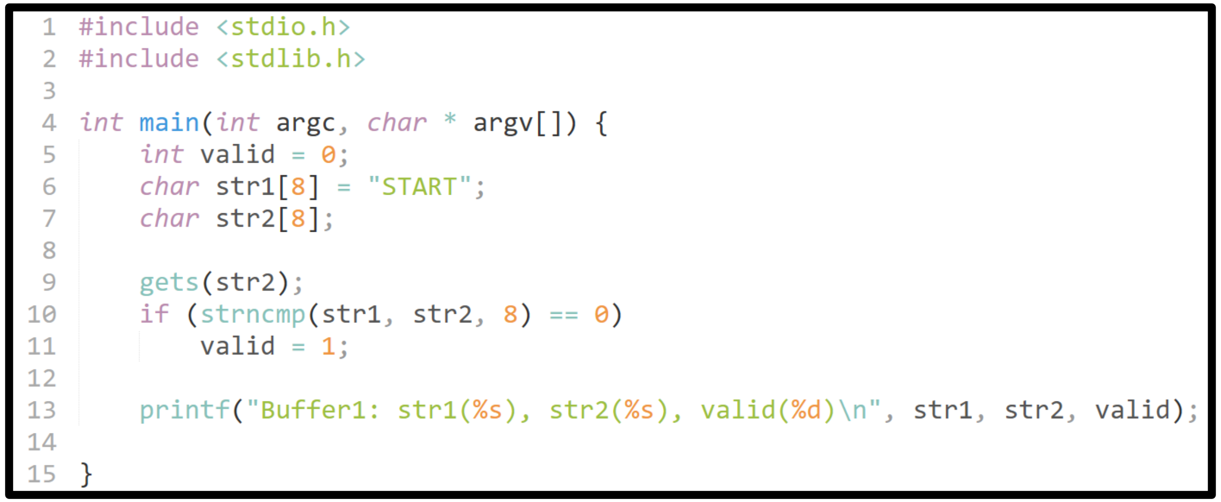
```
seunghoonwoo@seunghoonwoo-virtual-machine:~$ ./overflow
START
Buffer1: str1(START), str2(START), valid(1)
```



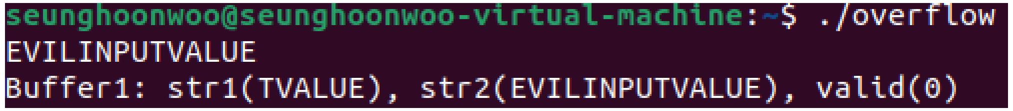
…		
0xffffd09c	S T A R	str2
0xffffd0a0	T 0 0 0	
0xffffd0a4	S T A R	str1
0xffffd0a8	T 0 0 0	
0xffffd0ac	0 0 0 1	valid

Memory safety

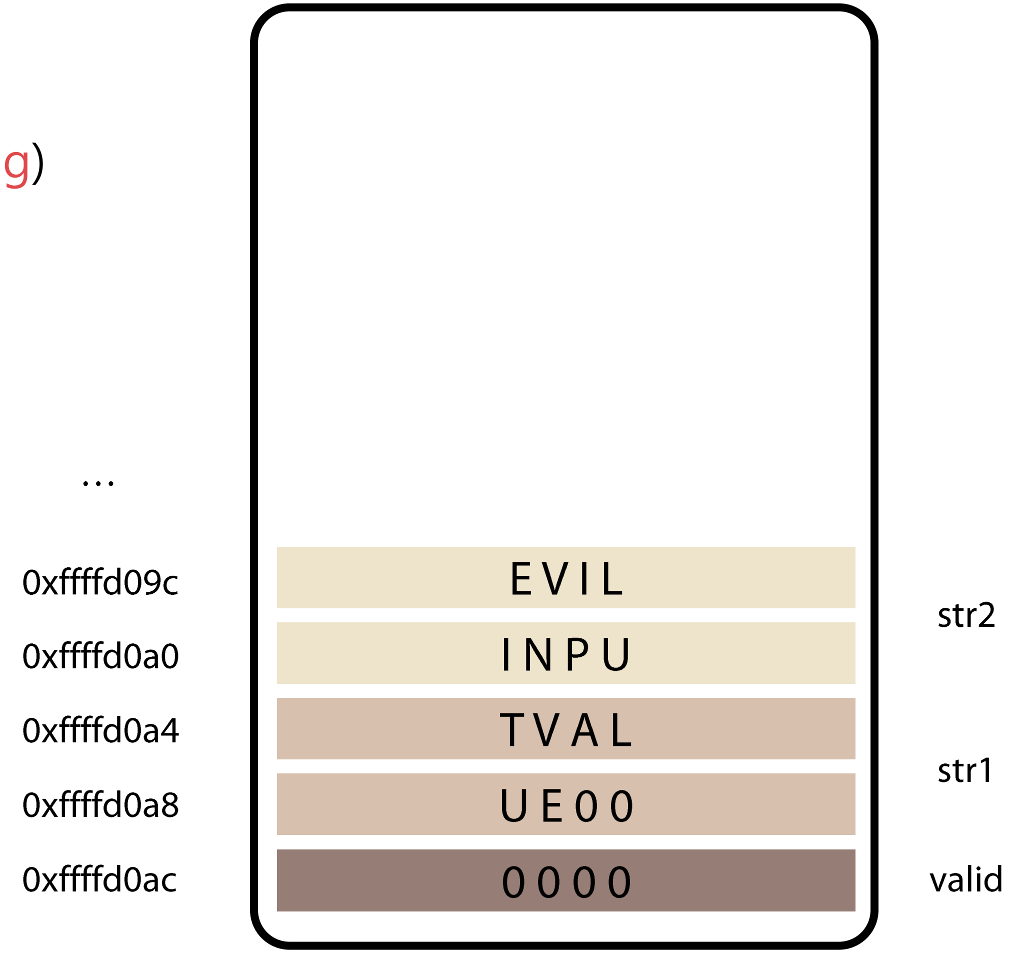
- **Stack buffer overflow** (something wrong)
  - Example: input = “EVILINPUTVALUE”



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

int main(int argc, char * argv[]) {
    int valid = 0;
    char str1[8] = "START";
    char str2[8];

    gets(str2);
    if (strncmp(str1, str2, 8) == 0)
        valid = 1;

    printf("Buffer1: str1(%s), str2(%s), valid(%d)\n", str1, str2, valid);

}
```



```
seunghoonwoo@seunghoonwoo-virtual-machine:~$ ./overflow
EVILINPUTVALUE
Buffer1: str1(TVALUE), str2(EVILINPUTVALUE), valid(0)
```



...

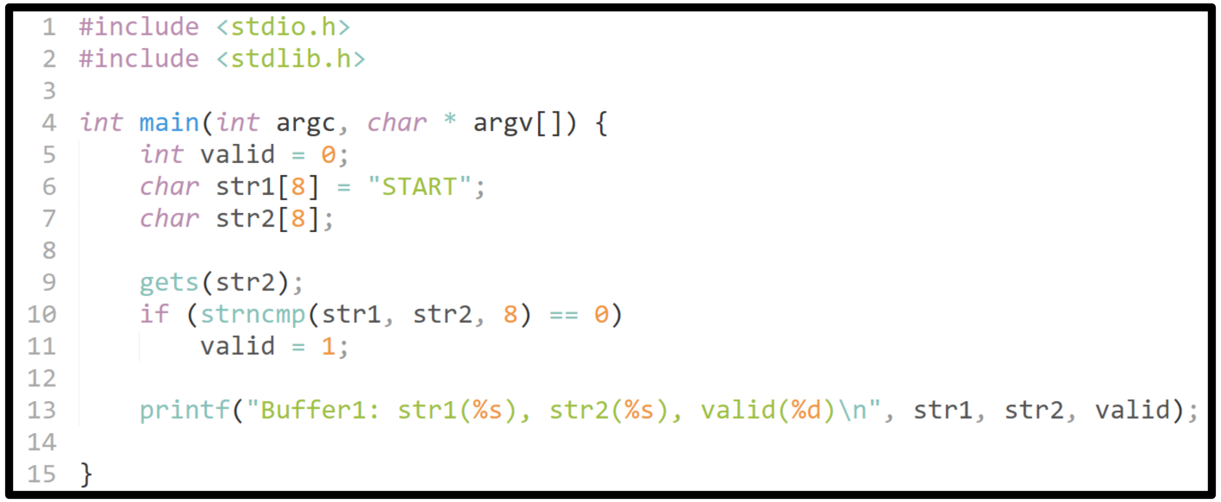
0xffffd09c EVIL
0xffffd0a0 INPU
str2

0xffffd0a4 TVAL
0xffffd0a8 UE00
str1

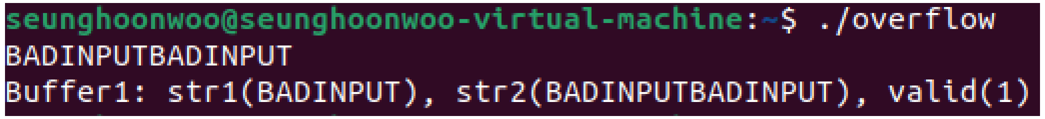
0xffffd0ac 0000 valid

Memory safety

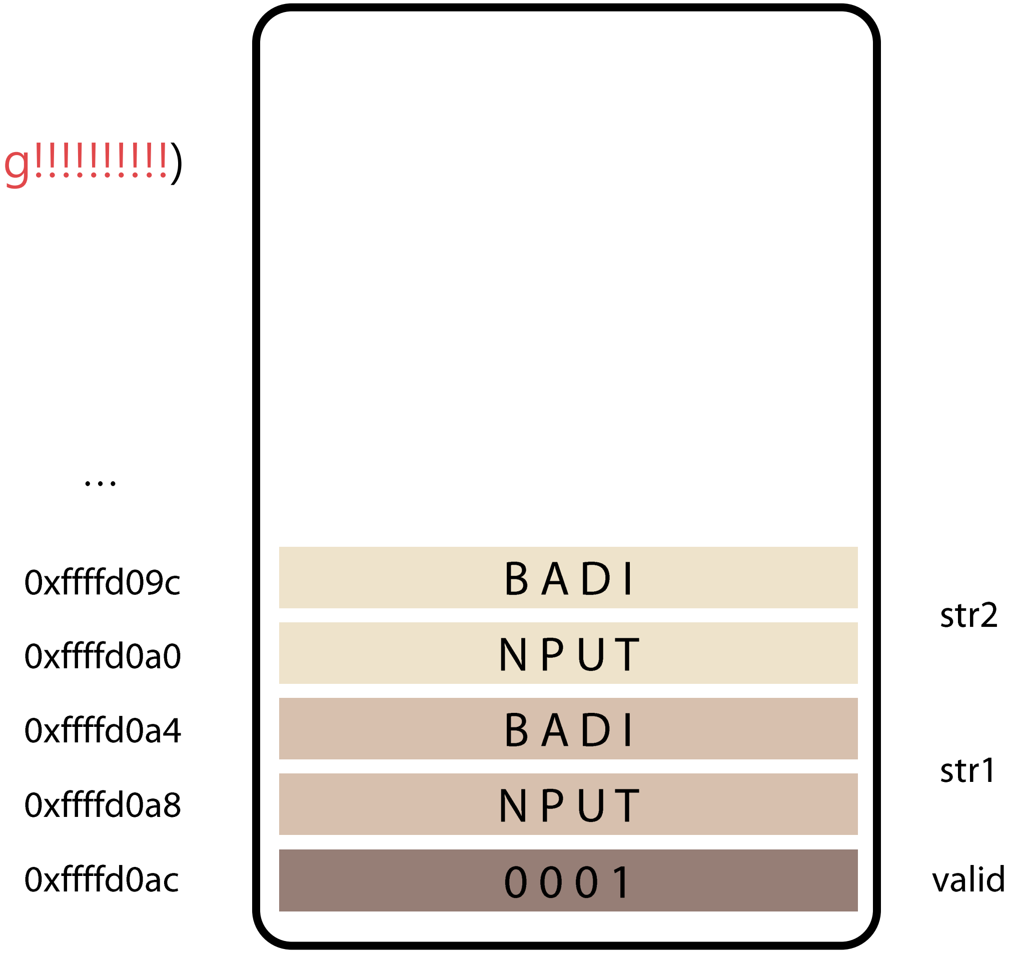
- **Stack buffer overflow** (critically wrong!!!!!!!!!!)
  - Example: input = “BADINPUTBADINPUT”



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

int main(int argc, char * argv[]) {
    int valid = 0;
    char str1[8] = "START";
    char str2[8];

    gets(str2);
    if (strncmp(str1, str2, 8) == 0)
        valid = 1;

    printf("Buffer1: str1(%s), str2(%s), valid(%d)\n", str1, str2, valid);

}
```



```
seunghoonwoo@seunghoonwoo-virtual-machine:~$ ./overflow
BADINPUTBADINPUT
Buffer1: str1(BADINPUT), str2(BADINPUTBADINPUT), valid(1)
```



…		
0xffffd09c	B A D I	str2
0xffffd0a0	N P U T	
0xffffd0a4	B A D I	str1
0xffffd0a8	N P U T	
0xffffd0ac	0 0 0 1	valid

Memory safety

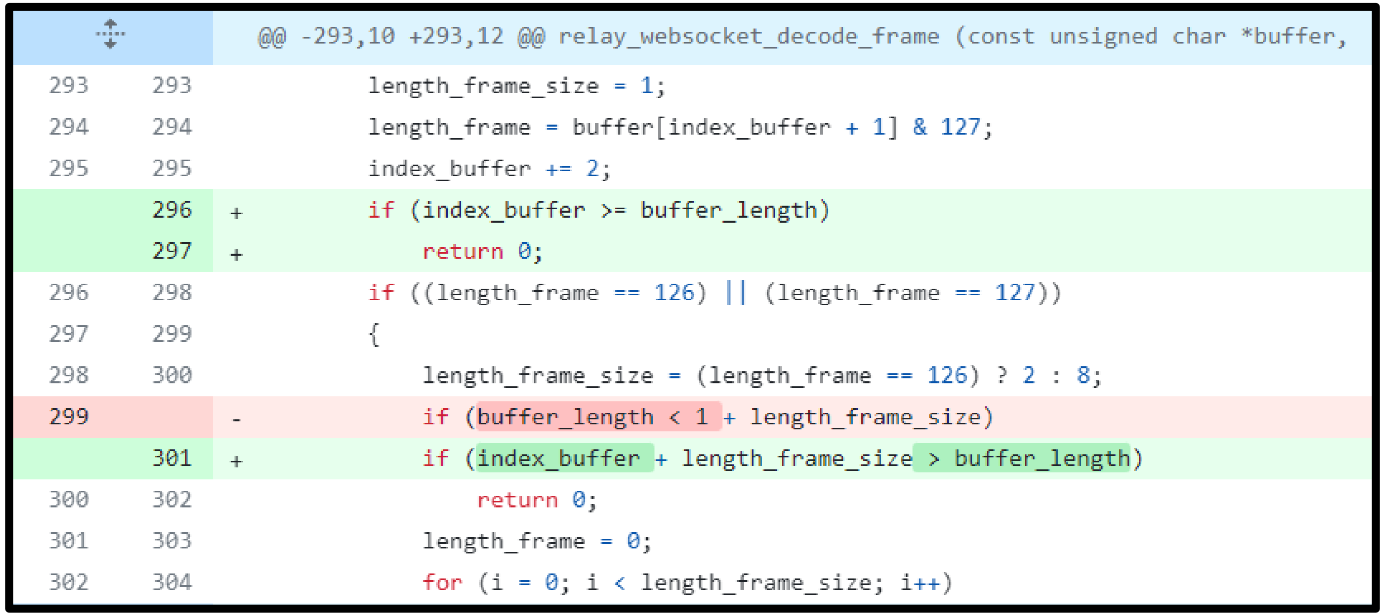
- **Stack buffer overflow**
    - Possible attack method
        - Manipulating RET value
            - RET: Memory address value where the command to be executed after the function ends
            - After saving the (malicious) code that executes the shell, if we write the memory address to the RET area, it will be executed after the function ends

Memory safety

- **Stack buffer overflow**
  - Related CWEs
    - CWE-121: Stack-based Buffer Overflow
    - CWE-131: Incorrect Calculation of Buffer Size

Memory safety

- **Stack buffer overflow**
  - Real-world example: WeeChat vulnerability (CVE-2021-40516)



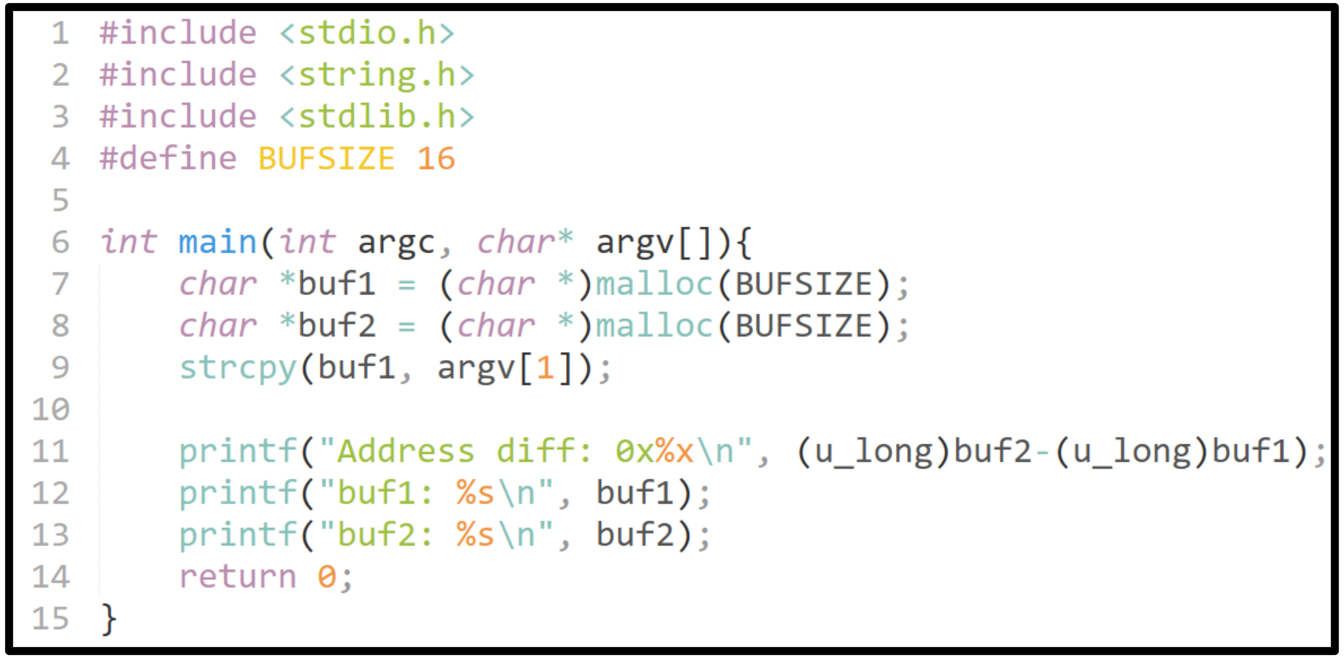
```
@@ -293,10 +293,12 @@ relay_websocket_decode_frame (const unsigned char *buffer,
          length_frame_size = 1;
          length_frame = buffer[index_buffer + 1] & 127;
          index_buffer += 2;
+         if (index_buffer >= buffer_length)
+             return 0;
          if ((length_frame == 126) || (length_frame == 127))
          {
              length_frame_size = (length_frame == 126) ? 2 : 8;
-             if (buffer_length < 1 + length_frame_size)
+             if (index_buffer + length_frame_size > buffer_length)
                  return 0;
              length_frame = 0;
              for (i = 0; i < length_frame_size; i++)
```

Memory safety

- **Heap buffer overflow**
  - Occurs when the memory in the heap area exceeds the specified range
  - Unlike the stack, the size of heap area cannot be determined at compile time
    - Dynamically allocated during the program's execution
  - More complicated than the stack-based buffer overflow

Memory safety

- **Heap buffer overflow**
  - Example



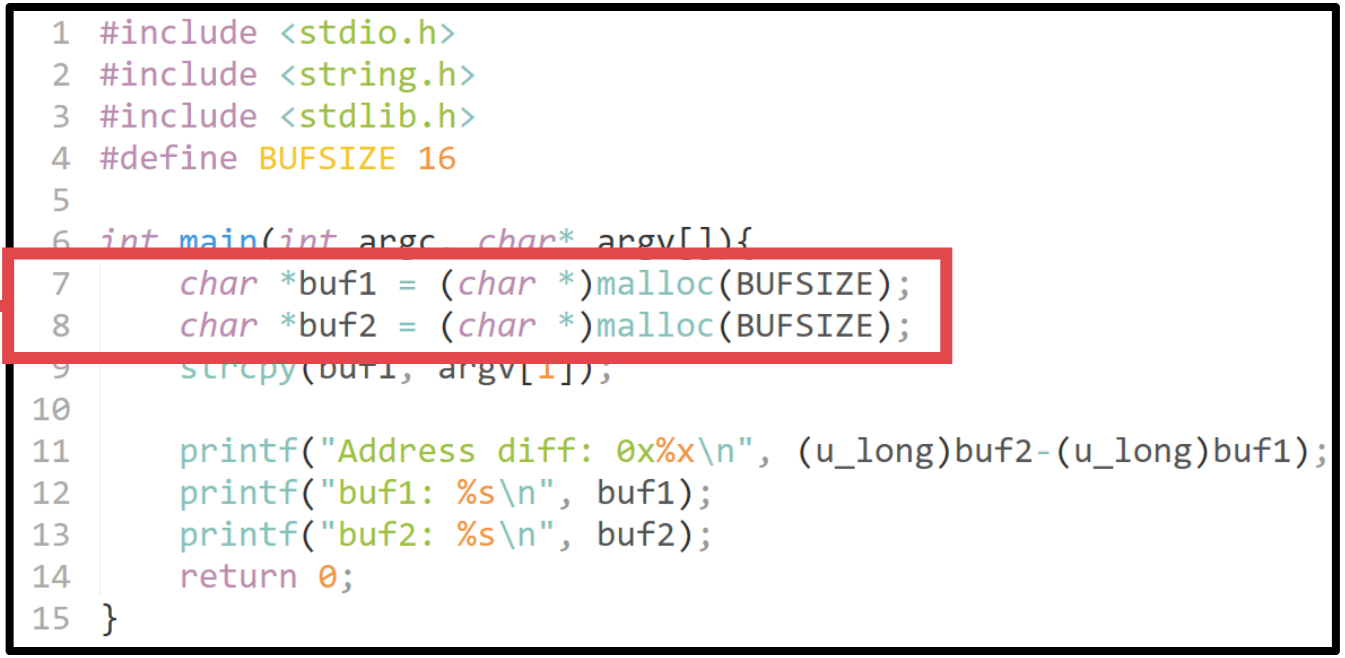
```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define BUFSIZE 16

int main(int argc, char* argv[]){
    char *buf1 = (char *)malloc(BUFSIZE);
    char *buf2 = (char *)malloc(BUFSIZE);
    strcpy(buf1, argv[1]);

    printf("Address diff: 0x%x\n", (u_long)buf2-(u_long)buf1);
    printf("buf1: %s\n", buf1);
    printf("buf2: %s\n", buf2);
    return 0;
}
```

https://itsaessak.tistory.com/114

Memory safety

- **Heap buffer overflow**
    - Example

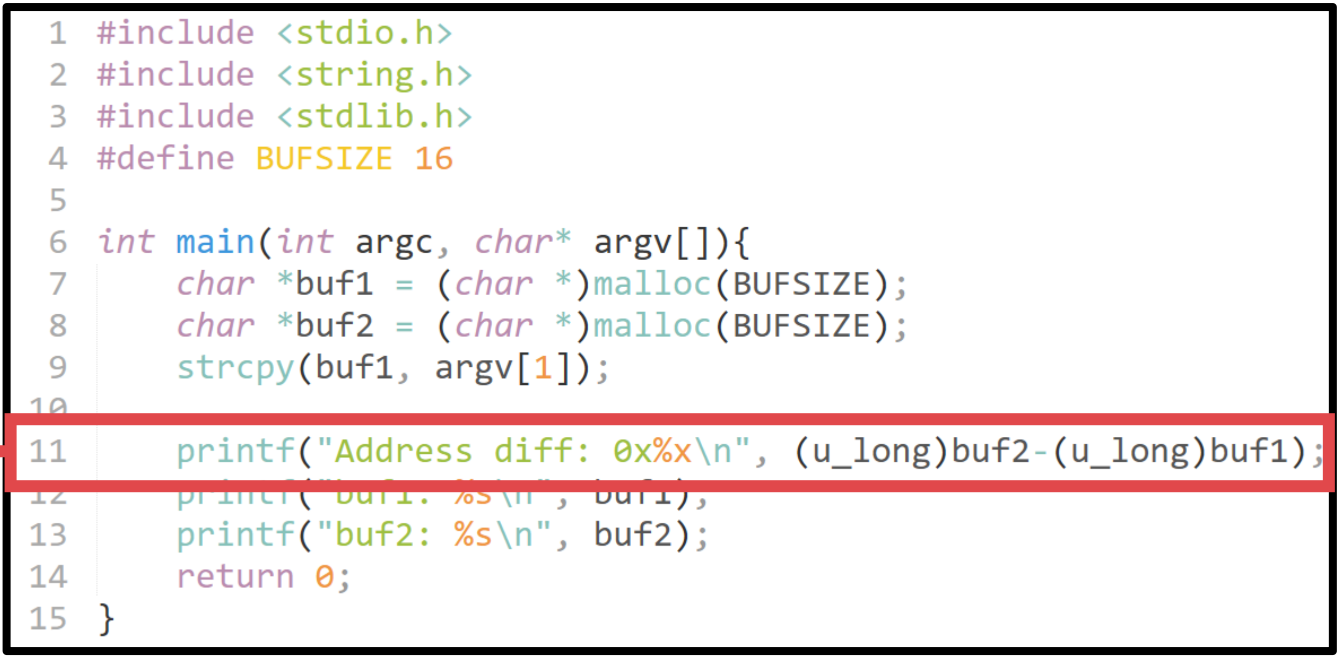
Dynamic allocation



```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define BUFSIZE 16

int main(int argc, char* argv[]){
    char *buf1 = (char *)malloc(BUFSIZE);
    char *buf2 = (char *)malloc(BUFSIZE);
    strcpy(buf1, argv[1]);

    printf("Address diff: 0x%x\n", (u_long)buf2-(u_long)buf1);
    printf("buf1: %s\n", buf1);
    printf("buf2: %s\n", buf2);
    return 0;
}
```

Memory safety

- **Heap buffer overflow**
  - Example

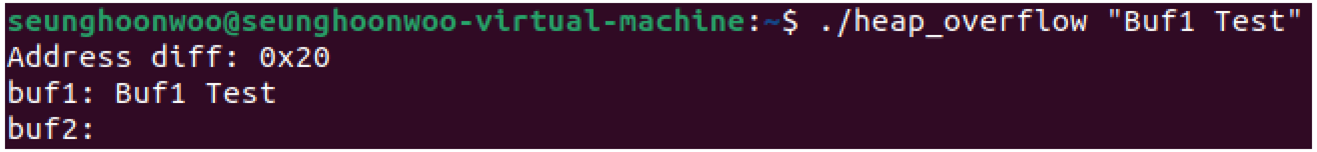


```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define BUFSIZE 16

int main(int argc, char* argv[]){
    char *buf1 = (char *)malloc(BUFSIZE);
    char *buf2 = (char *)malloc(BUFSIZE);
    strcpy(buf1, argv[1]);

    printf("Address diff: 0x%x\n", (u_long)buf2-(u_long)buf1);
    printf("buf1: %s\n", buf1);
    printf("buf2: %s\n", buf2);
    return 0;
}
```

The address diff between buf1 and buf2 for testing heap buffer overflow (this value can be found through GDB)

Memory safety

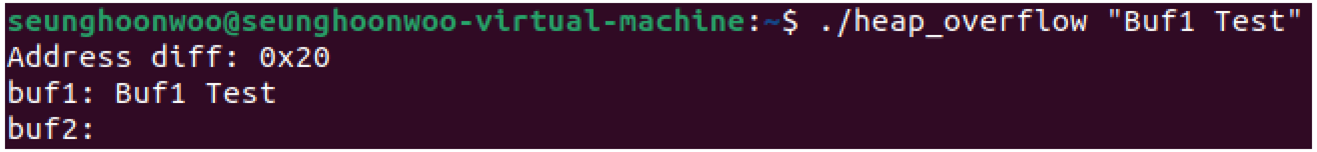
- **Heap buffer overflow**
  - Example



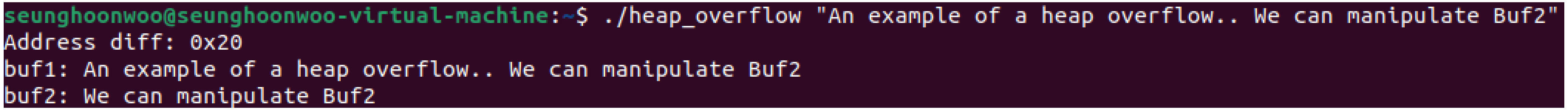
```
seunghoonwoo@seunghoonwoo-virtual-machine:~$ ./heap_overflow "Buf1 Test"
Address diff: 0x20
buf1: Buf1 Test
buf2:
```

Memory safety

- **Heap buffer overflow**
  - Example



```
seunghoonwoo@seunghoonwoo-virtual-machine:~$ ./heap_overflow "Buf1 Test"
Address diff: 0x20
buf1: Buf1 Test
buf2:
```



```
seunghoonwoo@seunghoonwoo-virtual-machine:~$ ./heap_overflow "An example of a heap overflow.. We can manipulate Buf2"
Address diff: 0x20
buf1: An example of a heap overflow.. We can manipulate Buf2
buf2: We can manipulate Buf2
```

Memory safety

- **Heap buffer overflow**
    - Related CWEs
        - CWE-122: Heap-based Buffer Overflow
        - CWE-131: Incorrect Calculation of Buffer Size

Next Lecture

- **Out-of-bounds (OOB) vulnerabilities**
- **Defense mechanism against buffer overflow**
- **Vulnerabilities caused by improper memory management**