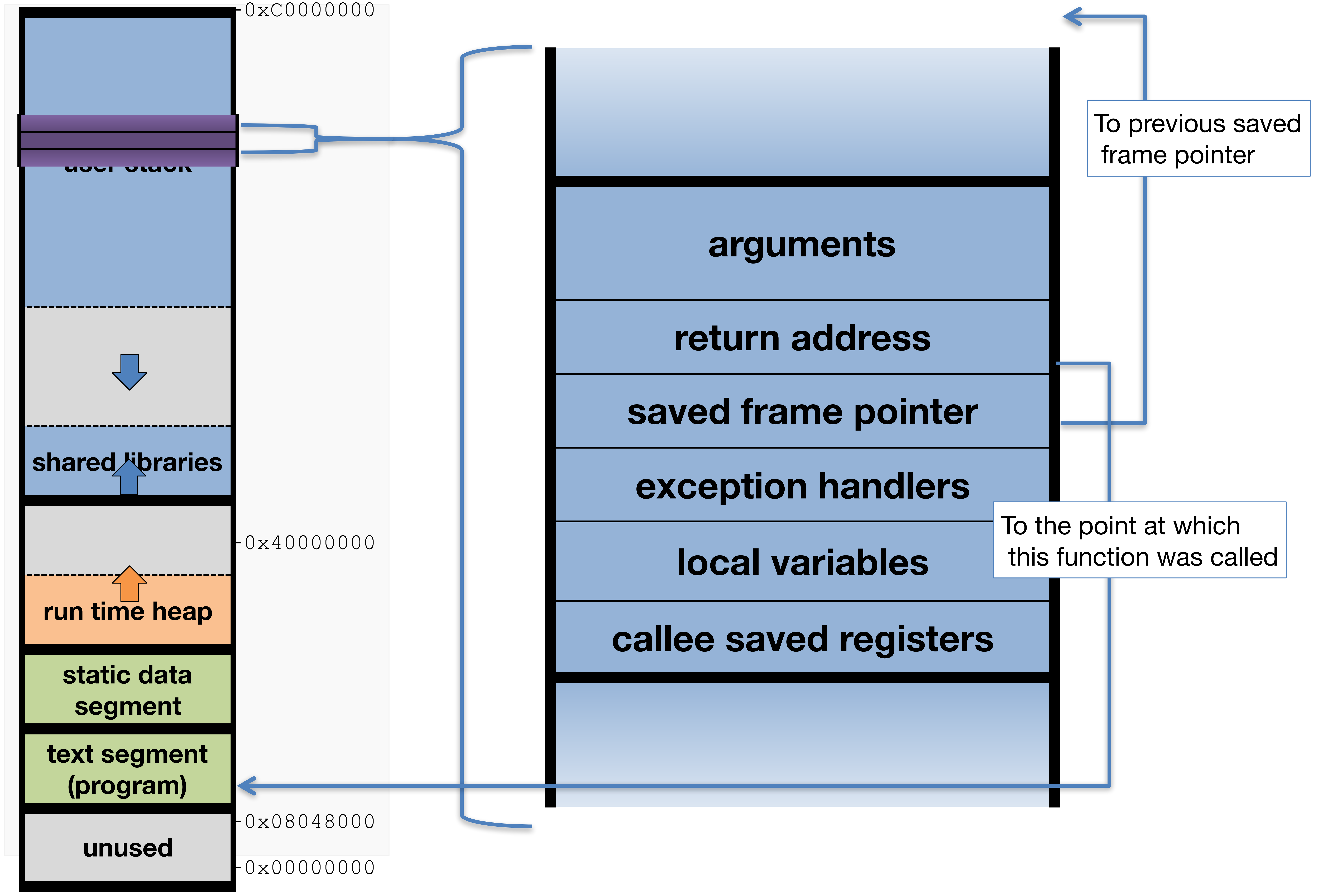
# Lecture 4: Memory Safety

https://cs161.org

## Announcements

- Homework 0 due Friday.
- Expect Homework 1 to be released later this week.

# Buffer Overflows


0xC0000000
user stack
shared libraries
0x40000000
run time heap
static data segment
text segment (program)
0x08048000
unused
0x00000000
arguments
return address
saved frame pointer
exception handlers
local variables
callee saved registers
To previous saved frame pointer
To the point at which this function was called

```
void safe() {
  char buf[64];
  ...
  fgets(buf, 64, stdin);
  ...
}
```

```
void safer() {
  char buf[64];
  ...
  fgets(buf, sizeof(buf), stdin);
  ...
}
```

Assume these are both under the control of an attacker.

```
void vulnerable(int len, char *data) {
  char buf[64];
  if (len > 64)
    return;
  memcpy(buf, data, len);
}
```

```
memcpy(void *s1, const void *s2, size_t n);
```

size_t is ***unsigned***:
What happens if len == -1?

```c
void safe(size_t len, char *data) {
  char buf[64];
  if (len > 64)
    return;
  memcpy(buf, data, len);
}
```

```
void f(size_t len, char *data) {
  char *buf = malloc(len+2);
  if (buf == NULL) return;
  memcpy(buf, data, len);
  buf[len] = '\n';
  buf[len+1] = '\0';
}
```

Is it safe?  Talk to your partner.

Vulnerable!
If **`len`** = **`0xffffffff`**, *allocates only 1 byte*

**Broward Vote-Counting Blunder Changes Amendment Result**

POSTED: 1:34 pm EST November 4, 2004

**BROWARD COUNTY, Fla. --** The Broward County Elections Department has egg on its face today after a computer glitch misreported a key amendment race, according to WPLG-TV in Miami.

Amendment 4, which would allow Miami-Dade and Broward counties to hold a future election to decide if slot machines should be allowed at racetracks, was thought to be tied. But now that a computer glitch for machines counting absentee ballots has been exposed, it turns out the amendment passed.

Broward County Mayor Ilene Lieberman says voting counting error is an "embarrassing mistake."

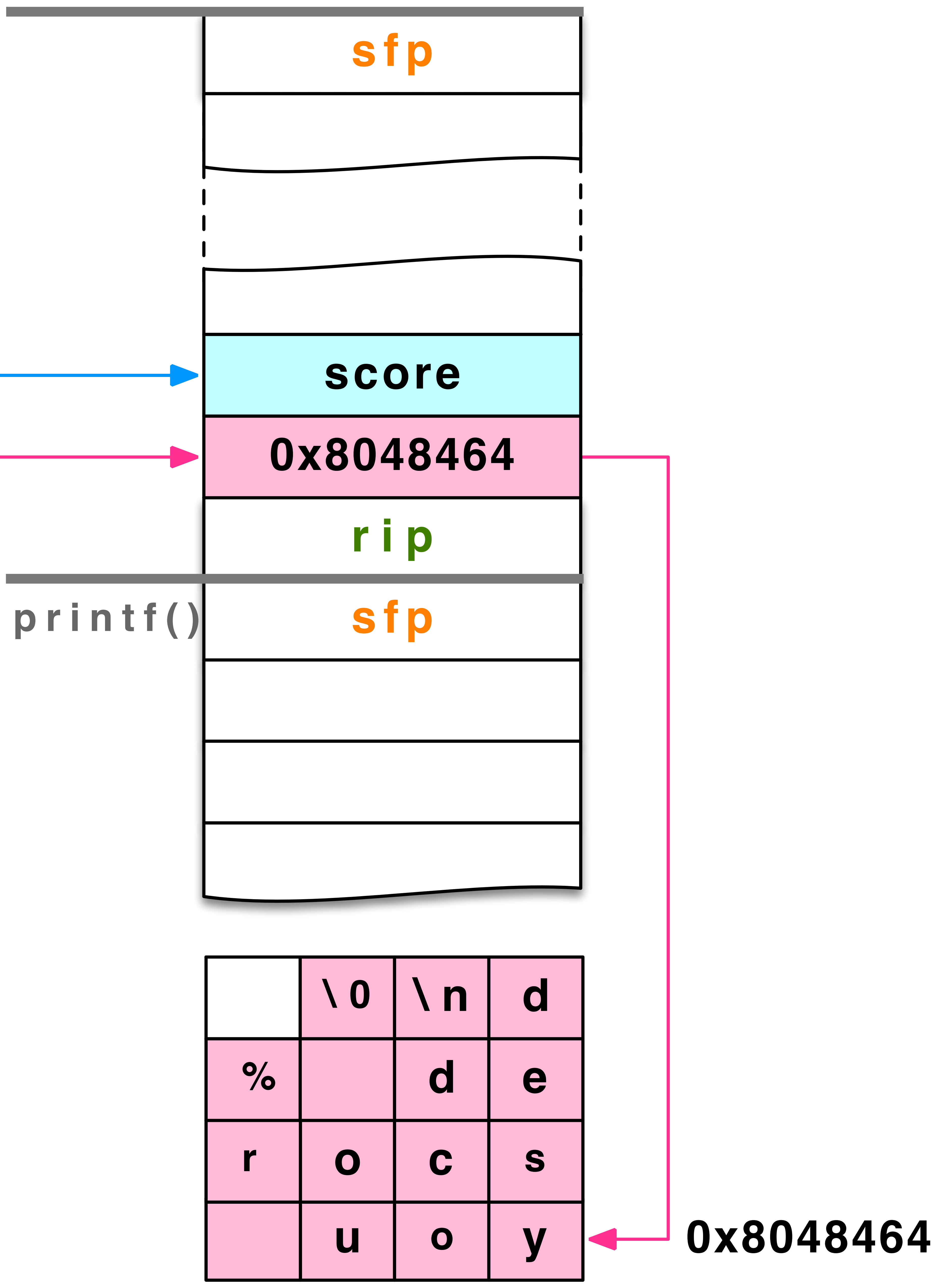
"The software is not geared to count more than 32,000 votes in a precinct. So what happens when it gets to 32,000 is the software starts counting backward," said Broward County Mayor Ilene Lieberman.

That means that Amendment 4 passed in Broward County by more than 240,000 votes rather than the 166,000-vote margin reported Wednesday night. That increase changes the overall statewide results in what had been a neck-and-neck race, one for which recounts had been going on today. But with news of Broward's error, it's clear amendment 4 passed.

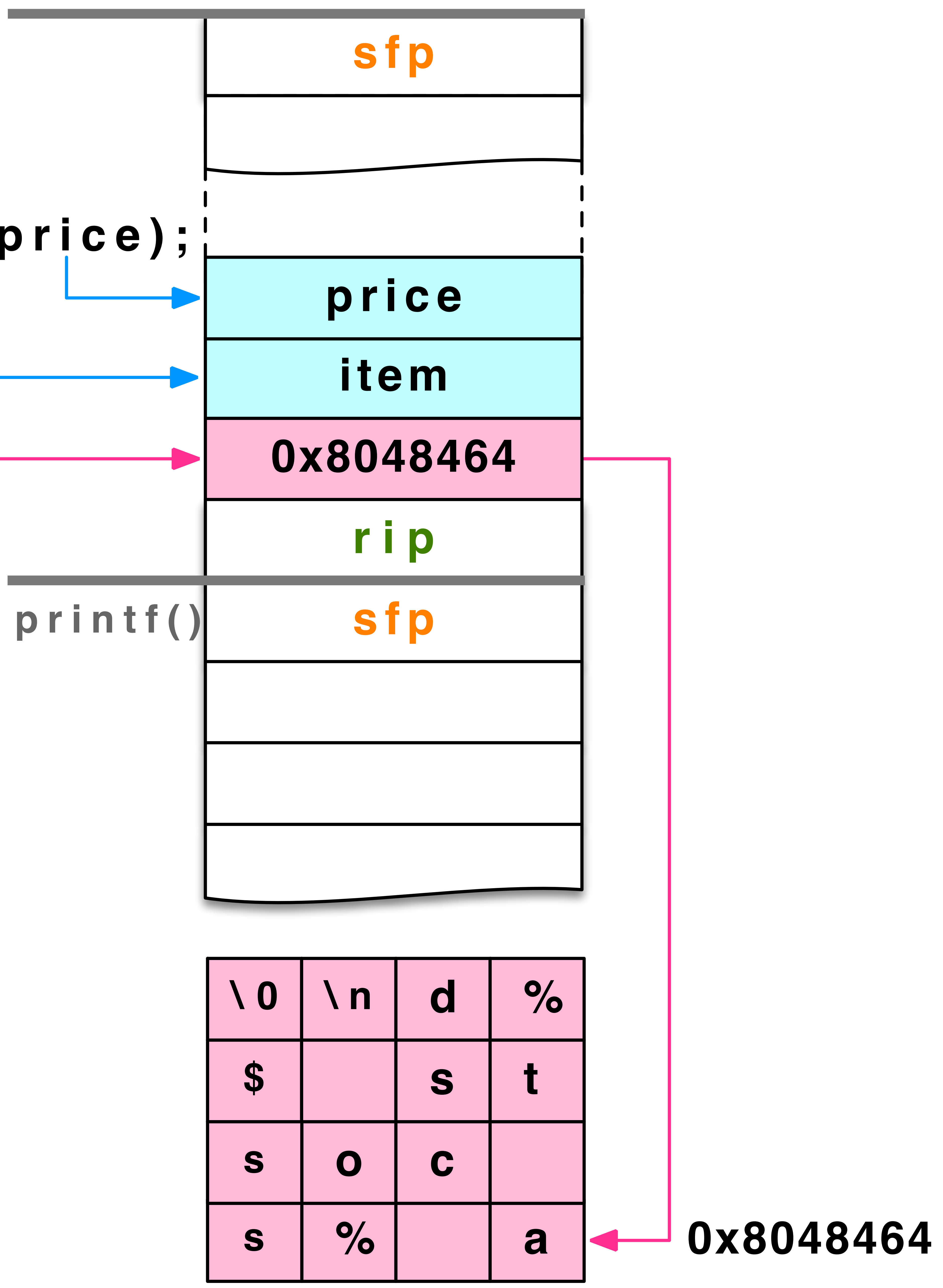
## Memory Safety

```c
void vulnerable() {
  char buf[64];
  if (fgets(buf, 64, stdin) == NULL)
    return;
  printf(buf);
}
```

```
printf("you scored %d\n", score);
printf("you scored %d\n", score);
```



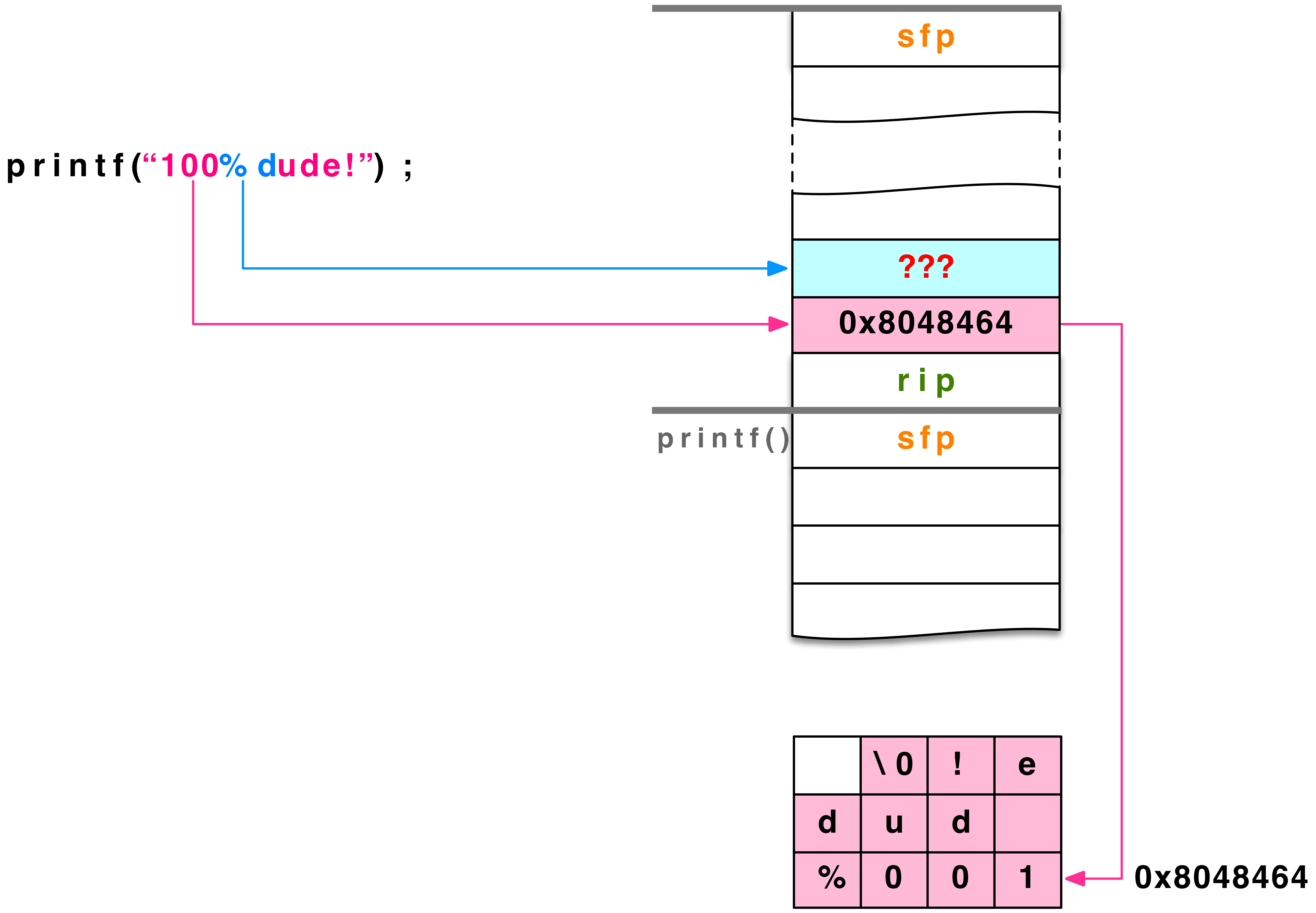
```
printf("a %s costs $%d\n", item, price);
printf("a %s costs $%d\n", item, price);
```



# Fun With **`printf`** format strings...

```
printf("100% dude!");
```

Format argument is missing!


printf("100% dude!") ;
sfp
???
0x8048464
rip
printf()
sfp
\0
!
e
d
u
d
%
0
0
1
0x8048464

# More Fun With `printf` format strings...

```
printf("100% dude!");
```

⇒ ***prints value 4 bytes above retaddr as integer***

```
printf("100% sir!");
```

⇒ ***prints bytes pointed to by that stack entry up through first NUL***

```
printf("%d %d %d %d ...");
```

⇒ ***prints series of stack entries as integers***

```
printf("%d %s");
```

⇒ ***prints value 8 bytes above retaddr plus bytes*** *pointed to by preceding stack entry*

```
printf("100% nuke'm!");
```

What does the %n format do??

%n *writes* the number of characters printed so far into the corresponding format argument.

```
int report_cost(int item_num, int price) {
  int colon_offset;
  printf("item %d:%n $%d\n", item_num,
                   &colon_offset, price);
  return colon_offset;
}
```

**`report_cost(3, 22)` prints `"item 3: $22"`**
**and returns the value 7**

**`report_cost(987, 5)` prints `"item 987: $5"`**
**and returns the value 9**

# Fun With `printf` format strings...

```
printf("100% dude!");
```

⇒ *prints value 4 bytes above retaddr as integer*

```
printf("100% sir!");
```

⇒ *prints bytes pointed to by that stack entry up through first NUL*

```
printf("%d %d %d %d ...");
```

⇒ *prints series of stack entries as integers*

```
printf("%d %s");
```

⇒ ***prints value 8 bytes above retaddr plus bytes** pointed to by preceding stack entry*

```
printf("100% nuke'm!");
```

⇒ ***writes the value 3 to the address pointed to by stack entry***

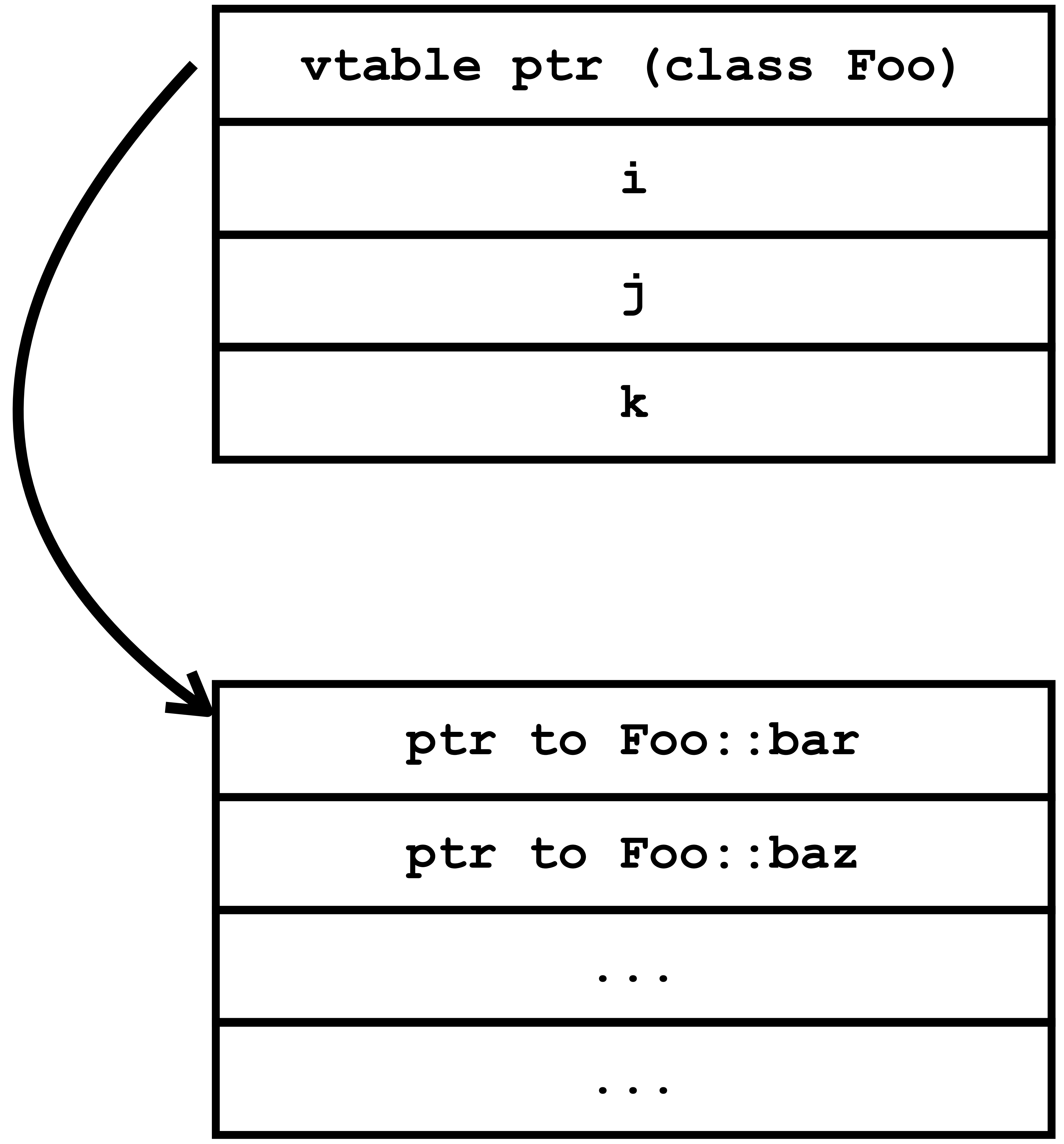
```
void safe() {
  char buf[64];
  if (fgets(buf, 64, stdin) == NULL)
    return;
  printf("%s", buf);
}
```

# It isn't just the stack...

- Control flow attacks require that the attacker overwrite a piece of memory that contains a pointer for future code execution
  - The return address on the stack is just the easiest target
- You can cause plenty of mayhem overwriting memory in the heap...
  - And it is made easier when targeting C++
- Allows alternate ways to hijack control flow of the program

# Compiler Operation: Compiling Object Oriented Code

```
class Foo {
   int i, j, k;
   public virtual void bar(){ ... }
   public virtual void baz(){ ... }
....
```



# A Few Exploit Techniques

- If you can overwrite a vtable pointer…
  - It is effectively the same as overwriting the return address pointer on the stack:
    When the function gets invoked the control flow is hijacked to point to the attacker's code
    - The only difference is that instead of overwriting with a pointer you overwrite it with a pointer to a table of pointers...
- Heap Overflow:
  - A buffer in the heap is not checked:
    Attacker writes beyond and overwrites the vtable pointer of the next object in memory
- Use-after-free:
  - An object is deallocated too early:
    Attacker writes new data in a newly reallocated block that overwrites the vtable pointer
  - Object is then invoked

# Magic Numbers & Exploitation…

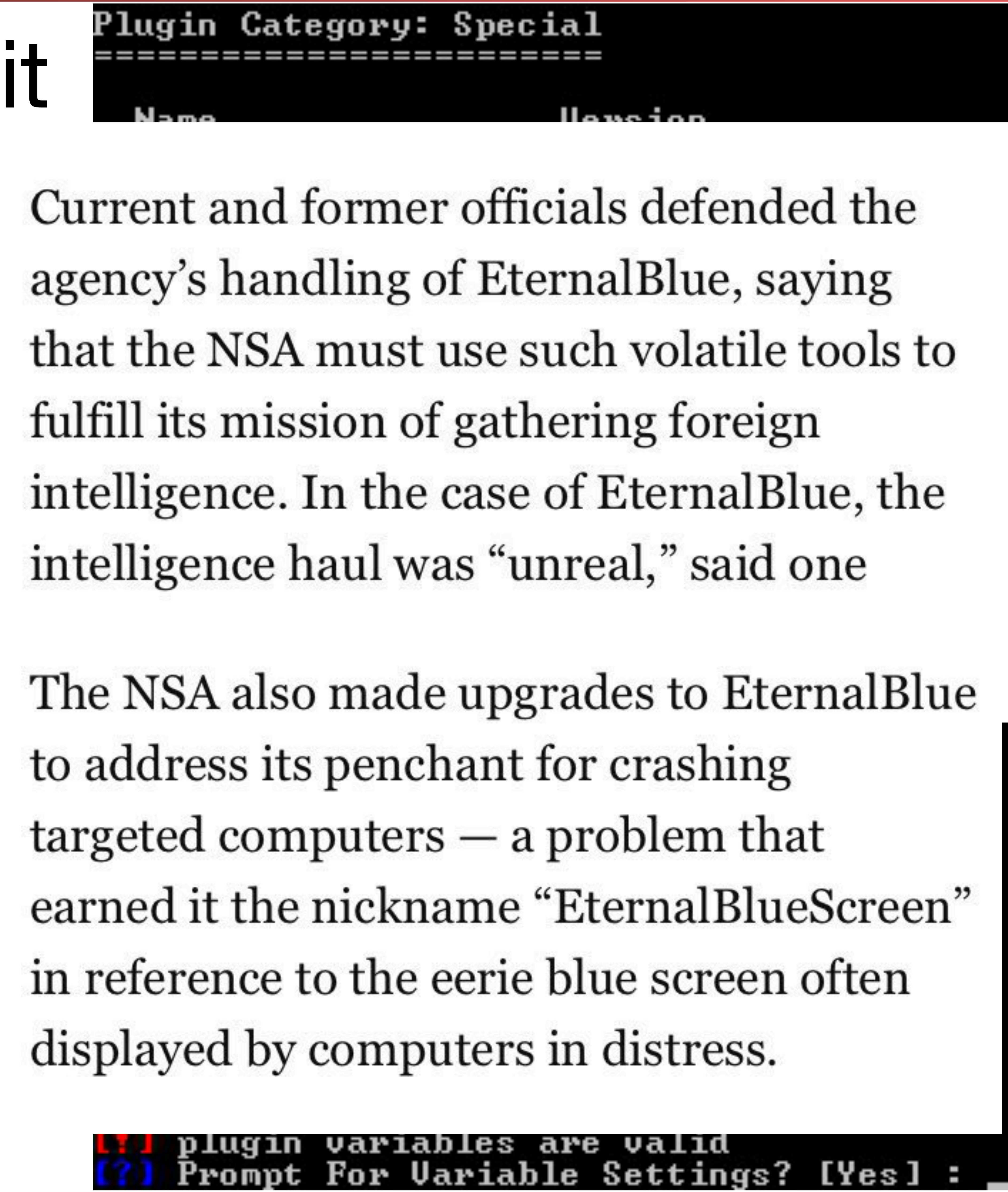
- Exploits can often be ***very*** brittle
  - You see this on your Project 1: Your ./egg will not work
    VM because the memory layout is different
- Making an exploit robust is an art unto itse
  - EXTRABACON is an NSA exploit for Cisco ASA “Adapt
    Appliances”
  - It had an exploitable stack-overflow vulnerability in the
  - But actual exploitation required two steps:
    Query for the particular version (with an SMTP read)
    Select the proper set of magic numbers for that versior

# A hack that helps: NOOP sled...

- Don't just overwrite the pointer and then provide the code you want to execute...
- Instead, write a large number of NOOP operations
  - Instructions that do nothing
- Now if you are a ***little*** off, it doesn't matter
  - Since if you are close enough, control flow will land in the sled and start running...

# ETERNALBLUE

- ETERNALBLUE is another NSA exploit
  - Stolen by the same group ("ShadowBrokers") v
  - Remote exploit for Windows through SMBv1 (V
- Eventually it was very robust...
  - But initially it was jokingly called ETERNALBLU crash Windows computers more reliably than e



```
Plugin Category: Special
========================

  Name                Version
```

Current and former officials defended the agency's handling of EternalBlue, saying that the NSA must use such volatile tools to fulfill its mission of gathering foreign intelligence. In the case of EternalBlue, the intelligence haul was "unreal," said one

The NSA also made upgrades to EternalBlue to address its penchant for crashing targeted computers — a problem that earned it the nickname "EternalBlueScreen" in reference to the eerie blue screen often displayed by computers in distress.

```
[!] plugin variables are valid
[?] Prompt For Variable Settings? [Yes] :
```

# Memory Safety

- Memory Safety: No accesses to undefined memory
  - "Undefined" is with respect to the semantics of the programming language
  - Read Access: attacker can read memory that he isn't supposed to
  - Write Access: attacker can write memory that she isn't supposed to
  - Execute Access: transfer control flow to memory they aren’t supposed to
- Spatial safety: No access out of bounds
- Temporal safety: No access before or after lifetime of object

## The CWE Top 25

Below is a brief listing of the weaknesses in the 2019 CWE Top 25, including the overall score of each.

| Rank | ID | Name | Score |
|---|---|---|---|
| [1] | CWE-119 | Improper Restriction of Operations within the Bounds of a Memory Buffer | 75.56 |
| [2] | CWE-79 | Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting') | 45.69 |
| [3] | CWE-20 | Improper Input Validation | 43.61 |
| [4] | CWE-200 | Information Exposure | 32.12 |
| [5] | CWE-125 | Out-of-bounds Read | 26.53 |
| [6] | CWE-89 | Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') | 24.54 |
| [7] | CWE-416 | Use After Free | 17.94 |
| [8] | CWE-190 | Integer Overflow or Wraparound | 17.35 |
| [9] | CWE-352 | Cross-Site Request Forgery (CSRF) | 15.54 |
| [10] | CWE-22 | Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal') | 14.10 |
| [11] | CWE-78 | Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection') | 11.47 |
| [12] | CWE-787 | Out-of-bounds Write | 11.08 |
| [13] | CWE-287 | Improper Authentication | 10.78 |
| [14] | CWE-476 | NULL Pointer Dereference | 9.74 |
| [15] | CWE-732 | Incorrect Permission Assignment for Critical Resource | 6.33 |
| [16] | CWE-434 | Unrestricted Upload of File with Dangerous Type | 5.50 |
| [17] | CWE-611 | Improper Restriction of XML External Entity Reference | 5.48 |
| [18] | CWE-94 | Improper Control of Generation of Code ('Code Injection') | 5.36 |
| [19] | CWE-798 | Use of Hard-coded Credentials | 5.12 |

# Reasoning About Safety

- How can we have ***confidence*** that our code executes in a safe (and correct, ideally) fashion?
- Approach: build up confidence on a function-by-function / module-by-module basis
- Modularity provides boundaries for our reasoning:
  - ***Preconditions***: what must hold for function to operate correctly
  - ***Postconditions***: what holds after function completes
- These basically describe a contract for using the module
- Notions also apply to individual statements (what must hold for correctness; what holds after execution)
  - Stmt #1's postcondition should logically imply Stmt #2's precondition
  - Invariants: conditions that always hold at a given point in a function (this particularly matters for loops)

```
int deref(int *p) {
    return *p;
}
```

*Precondition*?

```
/* requires: p != NULL
              (and p a valid pointer) */
int deref(int *p) {
    return *p;
}
```

***Precondition***: what needs to hold for function to operate correctly.

Needs to be expressed in a way that a *person* writing code to call the function knows how to evaluate.

```
void *mymalloc(size_t n) {
    void *p = malloc(n);
    if (!p) { perror("malloc"); exit(1); }
    return p;
}
```

***Postcondition***?

```
/* ensures: retval != NULL (and a valid pointer) */
void *mymalloc(size_t n) {
    void *p = malloc(n);
    if (!p) { perror("malloc"); exit(1); }
    return p;
}
```

***Postcondition***: what the function promises will hold upon its return.

Likewise, expressed in a way that a person using the call in their code knows how to make use of.

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    total += a[i];
  return total;
}
```

***Precondition*?**

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:

(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:

(1) Identify each point of memory access?
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:

(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* ?? */
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:

1. Identify each point of memory access
2. Write down precondition it requires?
3. Propagate requirement up to beginning of function

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: a != NULL &&
                 0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

*size*(X) = number of *elements* allocated for region pointed to by X
*size*(NULL) = 0

This is an <u>abstract</u> notion, *not* something built into C (like **`sizeof`**).

Gene

(1) Id

(2) Write down precondition it requires

(3) Propagate requirement up to beginning of function

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: a != NULL &&
                 0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:

(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function?

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: a != NULL &&
                 0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

Let’s simplify, given that `a` never changes.

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: 0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: 0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:

(1) Identify each point of memory access

(2) Write down precondition it requires

(3) Propagate requirement up to beginning of function?

```
/* requires: a != NULL */
int sum(int a[], size_t n) {      ?
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: 0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:

(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function?

```
/* requires: a != NULL */
int sum(int a[], size_t n) {    ✓
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: 0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:

(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function?

```
/* requires: a != NULL */
int sum(int a[], size_t n) {    ✓
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: 0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

The `0 <= i` part is clear, so let’s focus for now on the rest.

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

?

General correctness proof strategy for memory safety:

(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function?

```
/* requires: a != NULL */
int sum(int a[], size_t n) {              ?
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* invariant?: i < n && n <= size(a) */
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:

(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function?

```
/* requires: a != NULL */
int sum(int a[], size_t n) {                ?
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* invariant?: i < n && n <= size(a) */
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

How to prove our candidate invariant?

`n <= size(a)` **is straightforward because** `n` **never changes.**

```c
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* invariant?: i < n && n <= size(a) */
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* invariant?: i < n && n <= size(a) */    ?
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

What about `i < n` ?

```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* invariant?: i < n && n <= size(a) */     ?
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

What about `i < n`? That follows from the loop condition.

```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* invariant: i < n && n <= size(a) */
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

At this point we know the proposed invariant will always hold...

```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* invariant: i < n && n <= size(a) */
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

… and we're done!

```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* invariant: a != NULL &&
       0 <= i && i < n && n <= size(a) */
    total += a[i];
  return total;
}
```

A more complicated loop might need us to use *induction*:

**Base case**: first entrance into loop.

**Induction**: show that *postcondition* of last statement of loop, plus loop test condition, implies invariant.

```c
int sumderef(int *a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += *(a[i]);
    return total;
}
```

```
/* requires: a != NULL &&
      size(a) >= n &&
            ???                    */
int sumderef(int *a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += *(a[i]);
    return total;
}
```

```
/* requires: a != NULL &&
      size(a) >= n &&
      for all j in 0..n-1, a[j] != NULL */
int sumderef(int *a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += *(a[i]);
    return total;
}
```

This may still be memory ***safe***
but it can still have undefined behavior!

```c
char *tbl[N]; /* N > 0, has type int */


int hash(char *s) {
  int h = 17;
  while (*s)
    h = 257*h + (*s++) + 3;
  return h % N;
}

bool search(char *s) {
  int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

```
char *tbl[N];

/* ensures: ??? */
int hash(char *s) {
  int h = 17;
  while (*s)
    h = 257*h + (*s++) + 3;
  return h % N;
}
```

What is the correct *postcondition* for hash()?
(a) 0 <= retval < N, (b) 0 <= retval,
(c) retval < N, (d) none of the above.
Discuss with a partner.

```
=0);
```

```
char *tbl[N];

/* ensures: ??? */
int hash(char *s) {
  int h = 17;
  while (*s)
    h = 257*h + (*s++) + 3;
  return h % N;
}
```

What is the correct *postcondition* for hash()?
(a) 0 <= retval < N, (b) 0 <= retval,
(c) retval < N, (d) none of the above.
Discuss with a partner.

```
=0);
```

```
char *tbl[N];

/* ensures: ??? */
int hash(char *s) {
  int h = 17;
  while (*s)
    h = 257*h + (*s++) + 3;
  return h % N;
}
```

What is the correct *postcondition* for hash()?
(a) 0 <= retval < N, (b) 0 <= retval,
(c) retval < N, (d) none of the above.
Discuss with a partner.

```
=0);
```

```
char *tbl[N];

/* ensures: ??? */
int hash(char *s) {
  int h = 17;
  while (*s)
    h = 257*h + (*s++) + 3;
  return h % N;
}
```

What is the correct *postcondition* for hash()?
(a) 0 <= retval < N, (b) 0 <= retval,
(c) retval < N, (d) none of the above.
Discuss with a partner.

`=0);`

```
char *tbl[N];

/* ensures: ??? */
int hash(char *s) {
  int h = 17;
  while (*s)
    h = 257*h + (*s++) + 3;
  return h % N;
}
```

What is the correct *postcondition* for hash()?
(a) 0 <= retval < N, (b) 0 <= retval,
(c) retval < N, (d) none of the above.
Discuss with a partner.

```
=0);
```

```
char *tbl[N];

/* ensures: 0 <= retval && retval < N */
int hash(char *s) {
  int h = 17;                      /* 0 <= h */
  while (*s)
    h = 257*h + (*s++) + 3;
  return h % N;
}

bool search(char *s) {
  int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

```
char *tbl[N];

/* ensures: 0 <= retval && retval < N */
int hash(char *s) {
  int h = 17;                         /* 0 <= h */
  while (*s)                          /* 0 <= h */
    h = 257*h + (*s++) + 3;
  return h % N;
}

bool search(char *s) {
  int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

```
char *tbl[N];

/* ensures: 0 <= retval && retval < N */
int hash(char *s) {
  int h = 17;                      /* 0 <= h */
  while (*s)                       /* 0 <= h */
    h = 257*h + (*s++) + 3;        /* 0 <= h */
  return h % N;
}

bool search(char *s) {
  int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

```
char *tbl[N];

/* ensures: 0 <= retval && retval < N */
int hash(char *s) {
  int h = 17;                      /* 0 <= h */
  while (*s)                       /* 0 <= h */
    h = 257*h + (*s++) + 3;        /* 0 <= h */
  return h % N; /* 0 <= retval < N */
}

bool search(char *s) {
  int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

```
char *tbl[N];

/* ensures: 0 ~~<= retval~~ && retval < N */
int hash(char *s) {
  int h = 17;                        /* 0 <= h */
  while (*s)                         /* 0 <= h */
    h = 257*h + (*s++) + 3;          /* 0 ~~<= h~~ */
  return h % N; /* 0 ~~<= retval~~ < N */
}

bool search(char *s) {
  int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

```
char *tbl[N];

/* ensures: 0 <= retval && retval < N */
unsigned int hash(char *s) {
  unsigned int h = 17;             /* 0 <= h */
  while (*s)                       /* 0 <= h */
    h = 257*h + (*s++) + 3;        /* 0 <= h */
  return h % N;          /* 0 <= retval < N */
}

bool search(char *s) {
  unsigned int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

# Memory safe languages

- Do you honestly think a human is going to go through this process for all their code?
  - Because that is what it takes to prevent undefined memory behavior in C or C++
- Instead, use a safe language:
  - Turns "undefined" memory references into an immediate exception or program termination
  - Now you simply don't have to worry about buffer overflows and similar vulnerabilities
- Plenty to chose from:
  - Python, Java, Go, Rust, Swift, C#, … Pretty much everything other than C/C++/Objective C