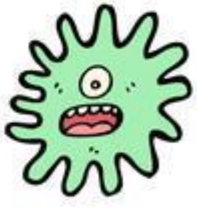
Memory corruption: detect and fix

Lecture 04.01

Case study by Scary Bug



Do we really need all these pointers?

- Pointers solve two common problems:
  - Allow different sections of code to share information without copying it
  - Enable complex "linked" data structures like linked lists and binary trees

What is the price we pay for using pointers?

- New and ugly types of bugs
- Random crashes
- Difficult to debug

5 common types of bugs

1. Using uninitialized memory: bad pointer
2. Trespassing on someone else's memory: overflow
3. Loosing pointer to memory storage: memory leak
4. Irresponsible function callers: memory leak
5. Accessing memory declared as free: dangling pointer

Recap: terminology

Referencing:

```
int i = 8;
int *ip;
ip = &i;
int **ipp;
ipp = &ip;
```

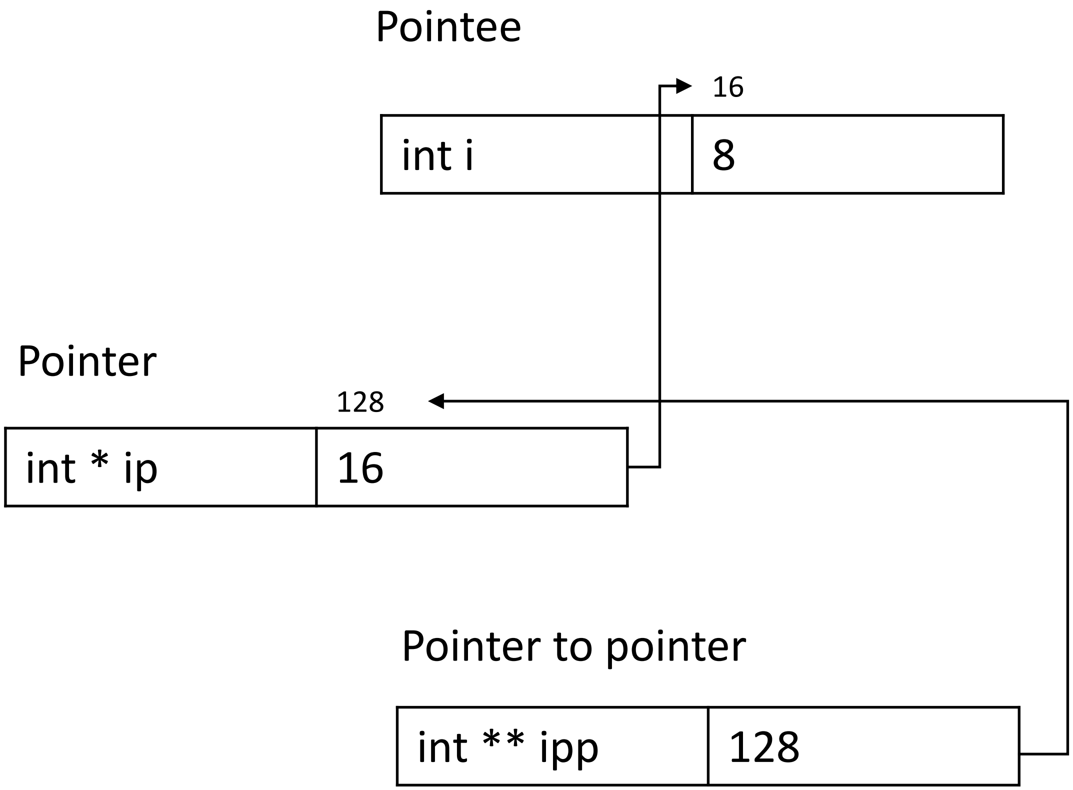
Dereferencing:

*ip is 8

**ipp is 8

*ipp = ip is 16

ipp is 128



Pointee

16

int i | 8

Pointer

128

int * ip | 16

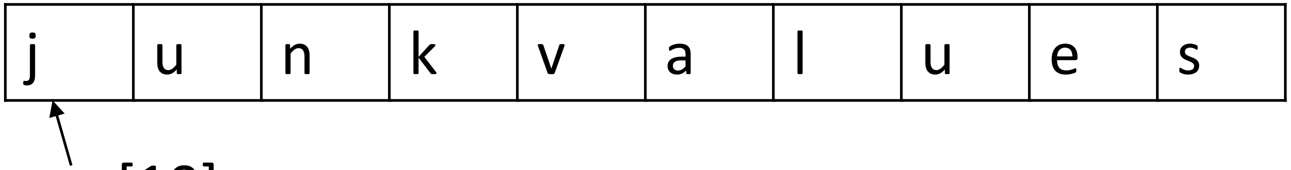
Pointer to pointer

int ** ipp | 128

Initialize your variables

Case of **bad pointers**

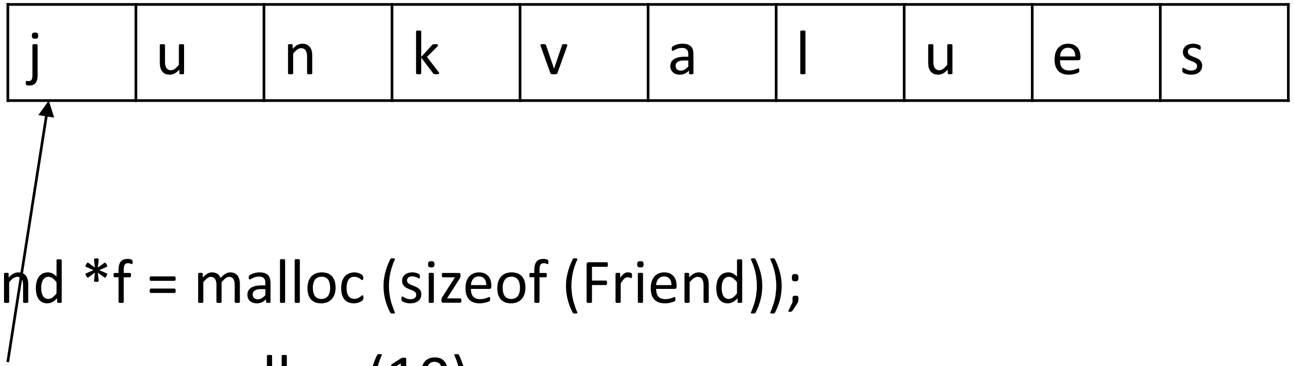
Freshly declared variables are initialized to:



j	u	n	k	v	a	l	u	e	s

```
char arr [10];
printf (“%s\n”, arr)
```

Freshly allocated memory contains:



j	u	n	k	v	a	l	u	e	s

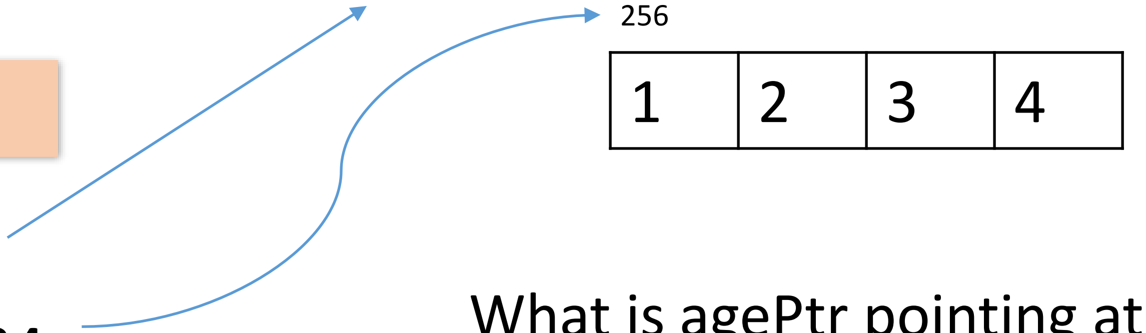
```
Friend *f = malloc (sizeof (Friend));
f->name = malloc (10);
```

Freshly declared pointer variable contains:

Junk address, say 256

BAD pointer

```
int * agePtr;
*agePtr = 1234;
```



1	2	3	4

What is agePtr pointing at? Whatever it is – we are writing 1234 into it

- We declare the int* (pointer to int) variable “agePtr"
- This allocates space for the pointer, but not the pointee!
- We access some random memory and destroy it

Example: bad pointer

```
int * agePtr;
*agePtr = 1234;
```

BAD pointer

Detect

Possible valgrind messages:

```
Use of uninitialised value
Conditional jump or move depends on uninitialised value
Segmentation fault (core dumped)
```

BAD pointer

Fix

```
int * agePtr = malloc (sizeof (int));
*agePtr = 22;
```

Exercise I. Detect and fix

```
// Returns a pointer to an int
int* killer() {
    int temp;
    return(&temp);
}

void victim() {
    int* ptr;
    ptr = killer();
    *ptr = 42;
}
```

BAD pointer

Pointer variables look like normal variables but require extra setup

- We are trained that when we allocate a simple variable, such as int, we can use it immediately
- Pointers look like other variables, but rules for their use are very different
- You have to assign your pointers to refer to valid pointees!

Don't be surprised when you forget!

BAD pointer

Because you *may* forget – make it a habit to set initial values to 0!

```
typedef struct person {
    char name [10];
    int age;
    struct person * next;
} Person;
```

- The sizeof each Person is 24, and there are bytes added for padding
- There is no way to set values of padding with a normal field assignment

BAD pointer

stdlib:
dynamic memory allocation

- malloc () - allocate some memory for us to use
- free () - release some memory that malloc() gave us earlier
- realloc () - change the size of some previously allocated memory
- calloc () - just like malloc(), except **clears the memory to zero**

calloc()

- calloc() is like malloc(), except that
  - **it clears the memory to zero for you**
  - it takes two parameters instead of one

```
p = malloc (10 * sizeof(int));
```

```
p = calloc (10, sizeof(int));
```

calloc()

- The pointer returned by calloc() can be used with realloc() and free() just as if you had used malloc()
- It takes time to clear memory
- But it is always safer to use calloc than make your pointer point to a random memory location!

NULL pointer

- NULL pointer is a pointer which is pointing to nothing.
- If we don’t have address to be assigned to a pointer, then we can simply use NULL
- **NULL vs Uninitialized pointer –** An uninitialized pointer stores an undefined value. A null pointer stores a defined value, but one that is defined by the environment to not be a valid address

Summary

- Set value of declared pointers to NULL
- Set values of variables to default values
- Use *malloc* + *memset*
- Use *calloc*

~~BAD pointer~~

Accessing memory that is used for something else

Case of **memory trespassing**

Truncate problem (from PCRS)

- Write a function named *truncate*() that takes a string *s* and a non-negative integer *n*.
- If *s* has more than *n* characters (not including the null terminator), the function should truncate *s* at *n* characters and return the number of characters that were chopped off.
- If *s* has *n* or fewer characters, *s* is unchanged and the function returns 0. For example, if s is the string "function" and n is 3, then truncate() changes s to the string "fun" and returns 5.

Solution on PCRS

```
int truncate (char *s, int n) {
    int retval = strlen(s);
    s[n] = '\0';
    retval -= strlen(s);

    return retval;
}
```

When will this code cause memory trespassing and how would you fix it?

Detect

Possible valgrind messages:

```
Invalid read of size 1
Invalid write
```

Trespassing

Fix

```
int truncate (char *s, int n) {
    int len = strlen(s);
    if (n <= len && n >=0) {
            s[n] = '\0';
            return len – n;
    }

    return 0;
}
```

Exercise II

```
void update_previous (char * str, int pos, char new_char) {
    int len = strlen (str);
    if (n <=len)
    str [n-1] = new_char;
}
```

Trespassing

Array safety principles

- *We need every occurrence of every subscript to be checked at run time against both the upper and the lower declared bounds of the array.*
- *People know how frequently subscript errors occur on production runs where failure to detect them could be disastrous.*
- *In any respectable branch of engineering, failure to observe such elementary precautions would have long been against the law.*

1980 Turing Award lecture, C. A. R. Hoare

Array bound checking in C: Variant 1

- If array was declared and initialized in this function:

```
for (int i=0; i< sizeof (array) / sizeof (array[0]); i++) {
   // Use array[i]
}
```

Problems with Variant 1

- The problem is with function calls, and passing your arrays as arguments: arrays decay to pointers when passed as an argument
- Thus the called function only sees a pointer to the first element

Array bound checking in C: Variant 2

- A common workaround is to simply have all your arrays terminated by a special character.
    - If you have an array of integers, then use INT_MAX (from *limits.h*) as the special value to signify end of the array
    - For all pointers use NULL (if NULL can legitimately appear in your array then use (NULL -1) instead)

Variant 2 example: array of ints

```
int array[] = {10, 12, 13, 14, 15, INT_MAX};
for (size_t i=0; array[i]!=INT_MAX; i++) {
  // Use array[i]
}
```

~~Trespassing~~

Variant 2 example: array of C strings

```
char *array[] = {
   "One", "Two", "Three", "Four", NULL
};

for (size_t i=0; array[i]; i++) {
   // Use array[i]
}
```

~~Trespassing~~

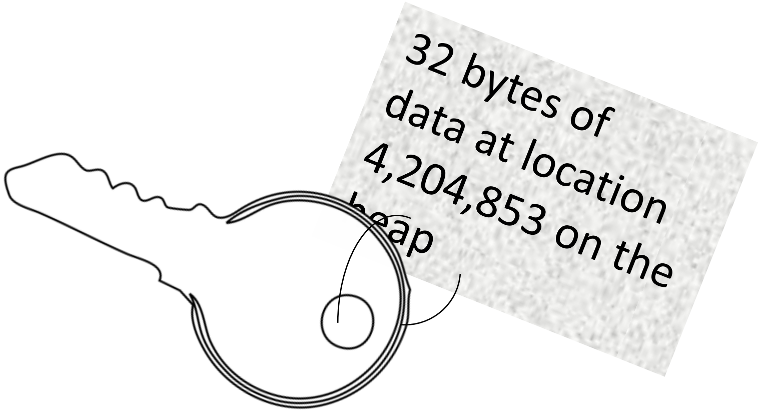
Summary

- Cross-check size of your arrays:
  both for upper and for lower bounds

~~Trespassing~~

Memory leak: orphan storage

Case of **lost keys**



malloc without *free*

- What happens if some memory is heap-allocated, but never deallocated?

- A program which forgets to deallocate a block is said to have a "*memory leak*"
- The heap gradually fills up, and blocks are not returned for re-use
- Memory leaks cause big problems for a program which runs for an indeterminate amount of time
- Many commercial programs have memory leaks

Memory leak

Example 1: pointer to memory block is lost

```
int * p = malloc (sizeof (int));
*p = 33;
int *s = malloc (sizeof (int));
*s = 22;
p = s;
```

Memory leak

Example 2: *free* Linked List

```
Person *head;
void free_person_list (Person *head) {
        Person *temp;
        Person *node = head;
        while (node != NULL) {
          temp = node;
          node = node->next;
          free (temp);
         }
  head = NULL;
}
```

```
typedef struct person {
   char * name ;
   struct person * next;
} Person;

Person * create_person (char *name) {
       p = malloc (sizeof (Person));
       p->name = malloc (strlen (name)+1);
       strcpy (p->name, name);
}
```

Memory leak

Detect

Possible valgrind messages:

```
X bytes in X blocks are definitely lost
More mallocs than frees
```

Memory leak

Fix

```
Person *head;
void free_person_list (Person *head) {
        Person *temp;
        Person *node = head;
        while (node != NULL) {
          temp = node;
          node = node->next;
          free (temp->name);
          free (temp);
         }
   head = NULL;
}
```

~~Memory leak~~

Exercise III

```
typedef struct person {
   char * name ;
   struct person * next;
} Person;

Person * create_person (char *name) {
       p = malloc (sizeof (Person));
       p->name = malloc (strlen (name)+1);
       strcpy (p->name, name);
}
```

```
void change_name (Person *p, char *new_name) {
       p->name = malloc (strlen (new_name)+1);
       strcpy (p->name, new_name);
}
```

Memory leak

realloc()

- Takes a chunk of memory you allocated with malloc() (or calloc()) and changes the size of the memory chunk
- Might have to move your data to another place in memory if it can't increase the size of the current block

- It means you should use realloc() sparingly since **it could be an expensive operation**
- Usually the procedure is to keep track of how much room you have in the memory block, and then add another big chunk to it if you run out

Example:
void add_data (int new_data)

```
// if data_count == data_size, the area is full and  needs to be realloc()'d
before we can add another:
    if (data_count == data_size) {
            // we're full up, so add a bucket
            data_size += BUCKET_SIZE;
            data = realloc (data, data_size * sizeof(int));
     }


    *(data+data_count) = new_data;
//  data[data_count] = new_data;
     data_count++;
```

Summary

- Free all dynamically allocated memory
- Check that number of mallocs is equal to number of frees
- Do not re-assign pointer without freeing its memory first: use *realloc* if you need to change the pointee

~~Memory leak~~

Someone needs to free the memory

Case of **irresponsible caller**

Example: return a copy of C string

```c
/* Given a C string, return a heap-allocated copy of the string.
The caller takes over ownership of the block and is responsible
for freeing it. */

char* StringCopy (const char* string) {
    char* newString;
    int len;
    len = strlen(string) + 1;
    newString = malloc(sizeof(char)*len);
    strcpy(newString, string);
    return (newString); // return a ptr to the block
}
```

Irresponsible call to StringCopy

```
int main () {
      char *copy1 = StrCopy (“one”);
      char *copy2 = StrCopy (“two”);
}
```

Irresponsible caller

Or even worse

```
int main () {
      StrCopy (“one”);
      StrCopy (“two”);
}
```

Irresponsible caller

Detect

Valgrind messages:

```
X bytes in X blocks are definitely lost
More mallocs than frees
```

Irresponsible caller

Two solutions for calling functions with dynamic memory requirements

Variant 1: Caller ownership

- The caller owns its own memory block. It may pass a pointer to this block to the callee for sharing purposes, but the caller retains ownership. The callee can access things in this block, and allocate and deallocate its own memory, but it should not move the caller's memory pointer

Variant 2: Callee allocated and returned

- The callee allocates some memory and returns it to the caller. The new memory is passed to the caller so they can see the result, and the caller must take over ownership of the memory

Variant 1 example: Caller owns the memory

```
void set_string (const char* string, char *new_value) {
      // copy the passed-in string to the passed-in block
      strcpy (string, new_value);
}


Int main () {
      char * copy1 = malloc (strlen (“one”)+1);
      set_string (copy1, “one”);
      …
      free  (copy1);
}
```

~~Irresponsible caller~~

Variant 2 example:
Callee allocated and returned

- Caller has to deallocate the memory after use

```
char * copy1 = StrCopy (“one”);
char *copy2 = StrCopy (“two”);
…

free (copy1);
free (copy2);
```

~~Irresponsible caller~~

Summary

- Each time you call a function that returns a pointer, the responsibility to free the memory is on the caller
- If you pass to the function a pre-allocated memory block – the responsibility to free it is on the caller

~~Irresponsible caller~~

Accessing memory after is was set free

Case of **dangling pointer**

Dangling pointer

free()

- This function takes as its argument a pointer that you've picked up using malloc() (or calloc())
- It releases the memory associated with that data

How does *free* work

- The program is finished using a block of memory, and it makes an explicit deallocation request to the heap manager
- The heap manager updates its private data structures to mark that block for future reuse
- Once you've freed some memory **you must remember not to use it any more**. Why?

```
free (p);
```

- After that, p still stores the same memory address. However this memory is declared ``available,'' and a later call to *malloc* might give that memory to some other part of your program
- By using the invalid pointer you will access and corrupt this new data

Dangling pointer

Example:
Dynamic multi-dimensional arrays

- Static 2D array:

```
char matrix[10][5]
```

- Dynamic array:

```
char ** strings;
```

Order of allocation

```
strings = (char **) malloc (10 * sizeof (char *));
for (i=0; i<10; i++)
     strings [i] = (char *)malloc (5 *sizeof (char));
```

Example: accessing pointer after free

```
free (strings);


for (i=0; i<10; i++)
      free (strings [i]);
```

Dangling pointer

Correct order of deallocation

```
for (i=0; i<10; i++)
      free (strings [i]);

free (strings);
```

~~Dangling pointer~~

To avoid problems with using p after free (p)

- **Always set p=NULL after call to free(p)!**

```
free (p);
…
p=NULL; //accessing this pointer accidentally will not harm some other newly allocated parts
…
free (p);  //freeing NULL pointer several times does not cause an error
```

~~Dangling pointer~~

Golden rules of C programming

- Initialize your memory. *Initialize your pointers*
- Cross-check *memory boundaries*
- *Free* pointee *memory* before moving its
- Preserve *correct order of memory deallocation* in nested structures
- Keep track of what was returned from a function and *free returned memory by the caller*
- Free dynamic memory after use. *Set pointer to NULL*