C Bootcamp

Day 3

CS3600, Spring 2013

Alan Mislove

Slides adapted from Anandha Gopalan’s CS132 course at Univ. of Pittsburgh and Pascal Meunier’s course at Purdue
Memory management
Memory management

Two different ways to look at memory allocation

- Transparent to user
  - Done by the compiler, OS
- User-defined
  - Memory allocated by the user

Why allocate memory?

- Required by program for permanent variables
- Reserve working space for program

Normally maintained using a stack and a heap
Stack

A stack is an organized data structure

Follows the LIFO (Last In First Out) principle

A data item which enters the stack last is the first to be taken out
Stack variables

```c
void foo(int i) {
    int j, k;
    char foo[] = "test";
}
```

Where are j, k, and foo allocated?

How long are they valid references? What is their scope?

```c
struct mystruct *foo(int i) {
    struct mystruct;
    mystruct.i = 7

    return &mystruct;
}
```

What happens if you try the program above?

Can the caller access mystruct?
Heap

A heap is a collection in memory manipulated by the user/system
No order specified for addition/removal
Dynamic memory allocation

There are times when we don’t know how much memory is required
- Employee data base doesn’t know how many employees will exist
- Program accepting command line input of different sizes
- User requests more records to be created

```c
printf("Add a new record y/n ?");
scanf("%c", &response);
if (response == 'y') {
    // we now have to allocate space
}
```

We use the heap to allocate memory dynamically
malloc

```c
void *malloc (size_t size)
```

Returns a pointer to a contiguous block in memory of `size` bytes

For example

```c
int *x = malloc(sizeof(int));
```

Returns a valid address, or NULL if an error occurs

*ALWAYS* check the return value of `malloc`

Memory is allocated (automatically) on the heap

Allocated space is not initialized; contains garbage

**Contained in the header file:** `<stdlib.h>` (usually) or `<alloc.h>`
A void *?

Why is `void *` returned?

- `malloc` just allocates memory which consists of some number of bytes.
- Memory as such has no data type.
- Programmer must associate a data type with the block of memory.

How to assign a data type to the memory just created?

Use type casting:

```c
double *x; // x is a pointer to double; malloc returns an address
x = (double *) malloc(100); // type casting the memory to be a double*
```

How many `doubles` can we store? 100/8 = 12

We can clearly see a pit falls of `malloc` here:

```c
x = (double *) malloc(100 * sizeof(double)); // better
```
Variants of malloc

```
void *calloc (size_t count, size_t size)
```

Allocated space is automatically initialized to zero
Same as `malloc`, but is contained in `<stdlib.h>`
Still have to cast void pointer to correct type

Advantages over `malloc`
Allocated memory is automatically initialized
`size` and `count` are separated, making it easier to use
Less prone to making errors on the correct amount and/or count required

```
int *x;
x = (int *) calloc (10, sizeof (int));
```
Freeing memory

You, the programmer, are in charge of freeing memory

Only dynamically allocated memory can be freed (i.e., via `malloc()` or `calloc()`)

```c
int x; // memory for x is statically assigned here, cannot be freed
```

When finished using dynamically allocated memory, free it

```c
void free (void *blk);
```

Good practice to avoid memory leaks.

- Growing loss of memory due to unreleased locations
- Could prevent program from running properly, due to lack of memory
- Can run out of memory
Using free()

```c
char *x;
x = (char *) malloc (10 * sizeof (char));
free (x);  // frees the memory just assigned

Never free a memory block that is not dynamically allocated.

```c
int *x;
free (x);  // error
```

Never double-free

```c
char *x = (char *) malloc (10 * sizeof (char));
free (x);
free (x);  // error
```

Never access freed memory

```c
char *x = (char *) malloc (10 * sizeof (char));
free (x);
strcpy(x, "welcome");  // error
```
Keep track of your pointers!

Dangling pointers

```c
char *x, *y;

x = (char *) malloc (10 * sizeof (int));
y = (char *) malloc (10 * sizeof (int));

x = y; // cannot access what was in x anymore
```

Can easily lead to memory leaks
Writing safe code in unsafe languages (e.g., C)
A few tips

C provides much less help to the programmer than others
  You can get yourself into trouble easily
  Much more liberal with types

Also, much harder to debug
  Memory is just an array of bytes
    Programmer has to assign meaning
  You can overwrite memory, making the source of problems

A few tips will make your experience in this class easier
Buffer/array overflows

```c
char b[10];
b[10] = x;
```

Array starts at index zero
- So [10] is 11th element
- One byte outside buffer was referenced
- Off-by-one errors are common and can be exploitable!

Real example:
```c
int get_request (int d, char buffer[], u_short len) {
    u_short i;
    for (i=0; i< len; i++) {
        ...
    }
    buffer[i] = '\0';
    return i;
}
```
What happens with an overflow?

If memory doesn't exist:
  Bus error

If memory protection denies access:
  Segmentation fault
  General protection fault

If access is allowed, memory next to the buffer can be accessed
  Can overwrite the heap, stack
  Worst of all options; won’t detect immediately
  Can compromise your program
Preventing buffer overflows

```c
void foo(int *array, int len);
```

Always pass array/buffer length with array
Don’t assume you know the length
Many library functions require you to do this already

```c
int foo[3];
for (i=0; i<=3; i++)
    foo[i] = 0;
```

Remember that last element of \( n \)-length array is \( n-1 \)
Can happen to the best programmers
Dealing with strings

Recall, strings in C are NULL-terminated char*s

Most C functions don't guarantee NULL-termination
No guarantee that the strings you get are properly NULL-terminated

Need to carefully read the function description to figure out

When it may not NULL-terminate a string
How to check for NULL-termination
Where to append a NULL character yourself
A few string functions

char * strncpy(char * dst, const char * src, size_t len);

- *len* is the maximum number of characters to copy
- *dst* is NULL-terminated only if less than *len* characters were copied!
- All calls to *strncpy* must be followed by a NULL-termination operation

int strlen(const char * str);

- What happens when you call *strlen* on an improperly terminated string?
  - *Strlen* scans until a null character is found
  - Can scan outside buffer if string is not null-terminated
  - *Strlen* is not safe to call unless you know string is NULL-terminated
char * strcpy(char * dst, const char * src);

How can you use strcpy safely?

Set the last character of src to NULL
   Even if string is shorter than the entire buffer
Do not check according to strlen(src)!

Check that the src is smaller than or equal to dst
   Or allocate dst to be at least equal to the size of src