
CPTS 121 L25
LAB 7: POINTERS, DEBUGGING, GITHUB

WHAT IS A POINTER?

- A memory address.

```
[> ./tmp
myNumber = 100
[&myNumber = 0x16f4d3478%
> cat tmp.c
#include <stdio.h>

int main(void) {
        int myNumber = 100;
        printf("myNumber = %d\n&myNumber = %p", myNumber, &myNumber);
        return 0;
}
```

LIFE CYCLE OF A DYNAMICALLY ALLOCATED OBJECT

1. malloc – it is allocated
2. (usage) – depends on program
3. free – it is released from memory

WHY ARE POINTERS USEFUL?

```
[> gcc -o tmp tmp.c ; ./tmp
myNumber = 100
&myNumber = 0x16f6c3478
sizeof(char*) = sizeof(void*) = 8
```

- They only take (minimum) 8 bytes.
- Allows us to pass by address instead of value.
- Allows us to return multiple types and values without creating a struct.
- We can modify the value directly.
- We do not need to return values (but may want to return success flags).

TYPICAL OPERATION ON FUNCTION CALL

- Call a function
- **Copy each argument to the callee**
- Do some processing
- [non-void funcs] Return some value and copy it to the desired value

FUNCTION CALL WITH POINTERS

- Call a function
- **Copy each pointer address to the callee**
- Do some processing
- [non-void funcs] Return some value and copy it to the desired value

DIFFICULTIES OF POINTERS

- The * operator has many purposes
- You can read in an area where you are not supposed to
  - Leads to possible data leaks
  - Could crash the program

```
int main(void) {
        int myNumber = 100;
        printf("myNumber = %d\n&myNumber = %p\n", myNumber, &myNumber);
        printf("sizeof(char*) = sizeof(void*) = %lu\n", sizeof(char*));

        int * myNumPtr = &myNumber; // Declare pointer type
        *myNumPtr = 10; // Dereference pointer (use/modify value)
        myNumber = 10 * 10; // Product

        return 0;
}
```

OPERATORS WITH POINTERS

- ++, --, +=, -=, *, +, -
  - Technically, anything with math, but only use the above operator
- ++, --, +=, -= will modify the pointer's value
  - If you not make a copy of the starting address, IT WILL BE LOST

NEXT PAGE DETAILS

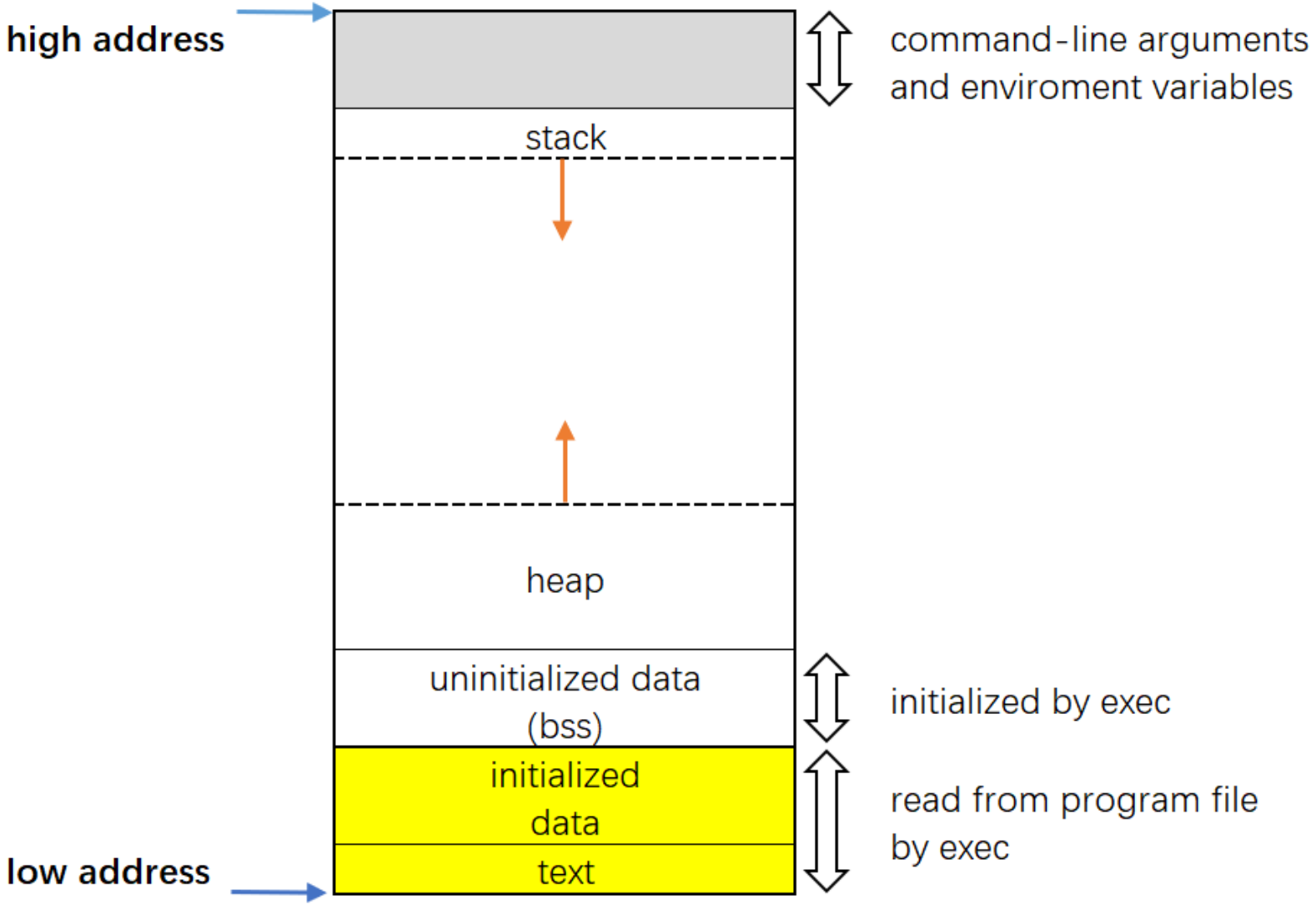
- C program on left is what outputs the top right screenshot
- Center right image shows how to count in Hex (w/ base 10 counterpart)
- Bottom right shows how the computer counts memory addresses when we use + operator.

```
[> cat tmp.c
#include <stdio.h>

#define N 0

void setIntToNum(int* ptr, int num) {
        *ptr = num;
        num = 250;
}

int main(void) {
        int myNumber = 100;
        printf("myNumber = %d\n&myNumber = %p\n", myNumber, &myNumbe
        printf("sizeof(char*) = sizeof(void*) = %lu\n", sizeof(char*

        int * myNumPtr = &myNumber; // Declare pointer type
        *myNumPtr = 10; // Dereference pointer (use/modify value)
        myNumber = 10 * 10; // Product

        // Assume myNumPtr is an array //
        myNumPtr++; // Go to the next int
        myNumPtr--; // Go to the last int
        myNumPtr += N; // Go to N ints away
        myNumPtr -= N; // Go to -N ints away
        (myNumPtr + N); // Look at int N away // myNumPtr is unmodified
        (myNumPtr - N); // Look at int -N away // myNumPtr is unmodified
        *myNumPtr = N; // Dereference and set myNumPtr to N

        int myPtrDst = 599;
        int myNum = 124;
        printf("myPtrDst = %d\tmyNum = %d\n", myPtrDst, myNum);
        setIntToNum(&myPtrDst, myNum);
        printf("myPtrDst = %d\tmyNum = %d\n", myPtrDst, myNum);

        printf("Observe memory addresses (Listed in order of declration):\n");
        printf("[int] myNumber = %p\n", &myNumber);
        printf("[int *] myNumPtr = %p\n", &myNumPtr);
        printf("[int] myPtrDst = %p\n", &myPtrDst);
        printf("[int] myNum = %p\n", &myNum);

        return 0;
}
>
```

```
[> ./tmp
myNumber = 100
&myNumber = 0x16bcd7478
sizeof(char*) = sizeof(void*) = 8
myPtrDst = 599  myNum = 124
myPtrDst = 124  myNum = 124
Observe memory addresses (Listed in order of declration):
[int] myNumber = 0x16bcd7478
[int *] myNumPtr = 0x16bcd7470
[int] myPtrDst = 0x16bcd746c
[int] myNum = 0x16bcd7468
>
```

```
0 = 0   4 = 4   8 = 8   12 = C
1 = 1   5 = 5   9 = 9   13 = D
2 = 2   6 = 6   10 = A  14 = E
3 = 3   7 = 7   11 = B  15 = F
>
```

```
sizeof(int) = 4
sizeof(int*) = 8
&x      = 0x16f7cb478
(&x)+1 = 0x16f7cb47c
(&x)+2 = 0x16f7cb480
(&x)+3 = 0x16f7cb484
Note: We are doing +1, but the system does
+4 since it knows the size of int is 4 bytes.
>
```



CWE – COMMON WEAKNESS ENUMERATION

- This is a community-driven list of common weaknesses in software
- The community submits common weaknesses and if accepted by other community members, is added to the list
- https://cwe.mitre.org/

CVE – COMMON VULNERABILITIES AND EXPOSURES

- A database where community members can submit vulnerabilities, but is not as likely to be posted.
- Look through if you are interested in cyber security.
- https://www.cve.org/

DEBUGGING

- Pause: Stop execution (resumable)
- Continue: Resume execution
- Stop: Terminate execution
- Step-over: Go to the next executable line
- Step-into: Step into the given function. (step-over if no function is available)
- Step-out: Go back to the caller, continue over remaining lines in current function

STEP OVER, STEP INTO

- Use step over when we want to go the next line
  - Typically to make sure values as expected and
  - To make sure the correct lines are executed (verify intended logic)
- Use step into when we want to view the function we call line by line
  - This will be very helpful with recursive functions
- Use step out when we have verified enough of the function and need to see it's use outside of the current function

CASE STUDY

```
1  nresp = packet_get_int();
2  if(nresp > 0)
3  {
4      response = xmalloc(nresp * sizeof(char*));
5      for(i = 0; i < nresp; i++)
6          response[i] = packet_get_string(NULL);
7  }
```

- Real code from OpenSSH Version < 3.4.
- Let us assume xmalloc is malloc.
- nresp is an unsigned integer.
- response is char ** (**2-D array of chars** or **an array of strings**)
- The problem with this code is nresp could be ≥ 1073741824. sizeof(char*) = 4.
  - When we multiply, we get 4,294,967,296 which is > 4,294,967,295 (max of unsigned int).
  - Now, only 1 byte is allocated by malloc and there is a buffer overflow.

More info/source: https://www.welivesecurity.com/2017/01/30/examples-vulnerable-code-find/

CASE STUDY (CONT.)

```
1  nresp = packet_get_int();
2  if(nresp > 0)
3  {
4      response = xmalloc(nresp * sizeof(char*));
5      for(i = 0; i < nresp; i++)
6          response[i] = packet_get_string(NULL);
7  }
```

- To Debug:
  1. Step over 1, look at the value of nresp. (Assume nresp = 1073741824)
  2. Step over 4, see if response was allocated.
     - It was, we see response is != NULL in our watch.
  3. Step over 5 (this is setting I = 0, checking if I < nresp then entering loop since its true
  4. Set response[i] to the result of packet_get_string
  5. Keep stepping through
     - We will eventually observe that the program crashes or is writing over data
     - This may not be completely evident on reading at response[0] (out of bounds)
       - It may take until response[100]

CASE STUDY (CONT.)

```
1  nresp = packet_get_int();
2  if(nresp > 0)
3  {
4      response = xmalloc(nresp * sizeof(char*));
5      for(i = 0; i < nresp; i++)
6          response[i] = packet_get_string(NULL);
7  }
```

- To make debugging easier
  - Move the (nresp * sizeof(char*)) expression to outside the xmalloc call and check the value.
    - Assume we save it to a variable called size.
    - We will see size = 0, which is not what we want.
    - Although size = 0, response will be given a memory address (non-NULL)
  - We should also check if response is non-null prior to doing the for loop
    - This code focuses on speed, so it does have debugging in mind.

GITHUB

- A common version control system (VCS)
- Helps us revert to previous functions in the case of corrupted files, messed up code, or other common issues you will encounter
- Helps us keep track of who changed certain parts of the project
- Allows global non-real time collaboration on projects
- Every time we make a change (or complete a function) we want to **commit**
  - This will create a version
- We want to **push** once we are done making local commits
- We want to **pull** in order to get data from the server

EXTENSIONS FOR VS

- **Code Alignment [Chris McGrath]**
- **SonarLint for Visual Studio (2017 | 2019 | 2022)**
- **Cppcheck add-in [Alexium]**
  - **Cppcheck includes checks for array bound overruns, unused functions in classes, uninitialized variables' memory/resource leaks and appropriate use of STL constructs. [1]**

[1] https://github.com/ajillepalli/HandsOnCyberTutorials/blob/main/StaticTestingStaticAnalysisandCodeReviews/StaticAnalysis&CodeReviews.pdf Page 10.

NAMING CONVENTIONS

- Macro constant, enum, union, static data member, global variable names: UPPERCASE_WITH_UNDERSCORE
- Functions: functionNameCamelCase()
- Variable names: variableWithCamelCase