Computer Systems & Low-Level Programming

C: pointers, call-by-value, call-by-reference, bit manipulation

Marija Stanojevic
Spring 2019

Review (1)

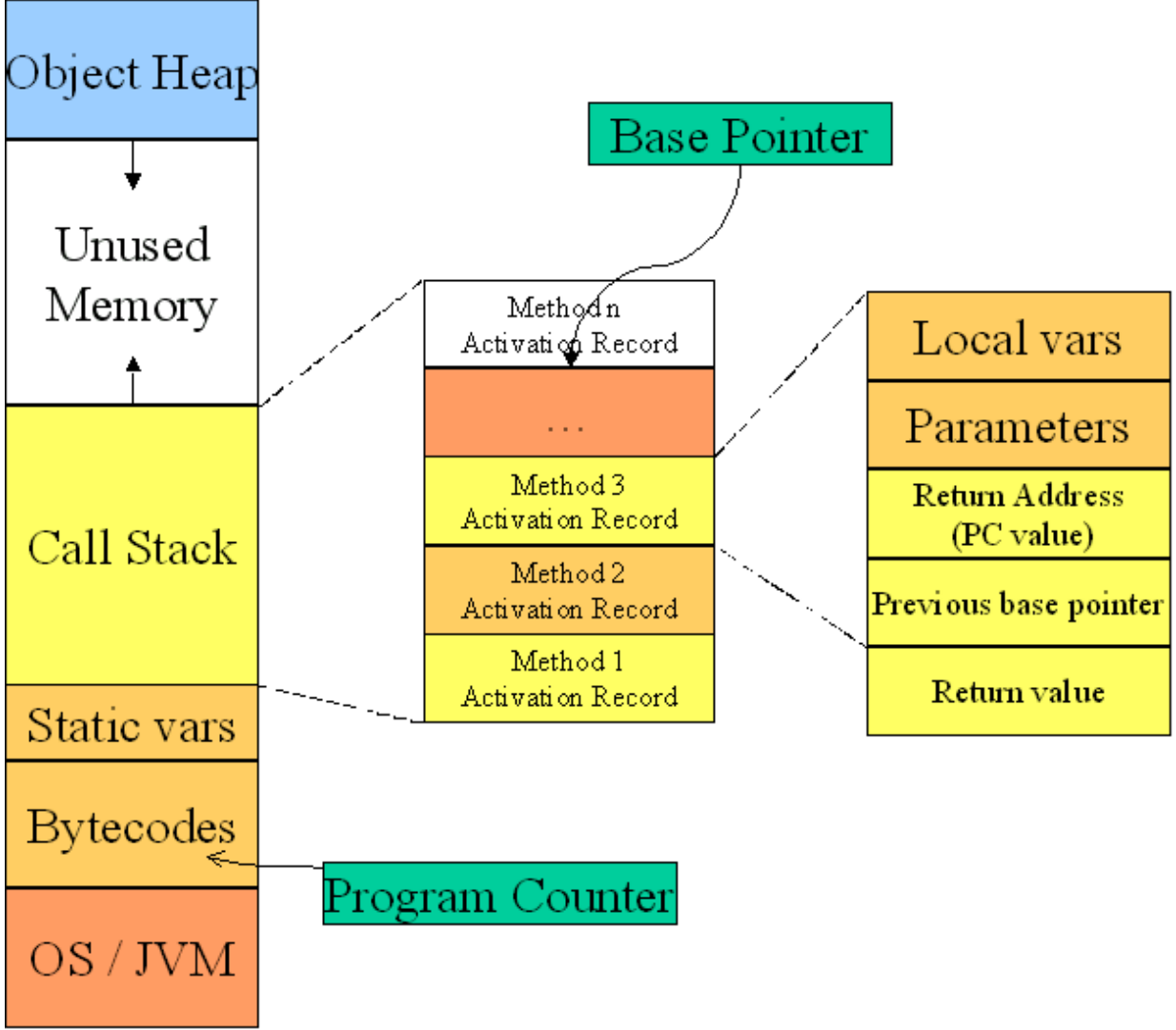
- Check comments you got for assignments
- Next assignments: if code doesn't run on server => 0 points for that work
- return 0; vs exit(0);
  - return leaves function and exit leaves the program; exit is defined in <stdlib.h>
- Division of two integers is integer
  - Cast one of the parameters to float in order to get float result (float)a/b; (float)5/2; => 2.5
  - (float) (a/b); would first calculate integer result and then turn it into float (float)(5/2); => 2.0
- Printing % => printf("Discount is 10%%"); would print: Discount is 10%
- Printing " => printf("This is \"Temple\""); would print: This is "Temple"
- scanf("%s", str); vs gets(str); vs fgets(str, 20, stdin);
  - scanf() and gets() allow don't check size of input, so they are considered unsafe
  - use fgets(s, 20, stdin) instead; second parameter is max size of input, last parameter says where to read input from (stdin - standard input or file pointer)

Review (2)

- Don’t specify format of printf in variable: ~~printf(str);~~ is wrong and insecure
- If you call function before you define it, you need to declare it on top of the file
  - float avg(int n);
- printf(“%.2f”, val); - prints float value with 2 decimal places
- break - in **loops** and **switch**
  - if you are in switch and use break you would exit from switch, but not from loop around it
  - if you want to break from loop under some condition, use if/else for condition
- Can’t condition on inequalities in **switch** in C
  - certain **case** is only one value, not a range of values; if you need ranges, use if/else
- After last character in string there must be ‘\0’ to announce C that string ends
  - Make sure to have char array of size at least n+1 for string of length n
- EOF character means that it’s end of file
  - if (c == EOF) printf(“File is read”);

Review (3)

- Variable naming rules in C:
  - use letters (uppercase and lowercase), numbers and _
  - can’t start with number
  - #define WORD_WORD 100; is most commonly used syntax to define symbolic constants
  - int length1 or int length_1 or strLength or str_length are all often used in community
  - Choose one style and stick to it in the whole code
- #define CON 100; vs const int con = 100;
  - first is symbolic constant and doesn’t occupy memory, instead every CON in code is substituted with 100. There is no datatype.
  - second is integer constant which is stored in memory as integer, but value can’t be changed
- sizeof (int) - gives the amount of bytes reserved in memory for that type
- fflush(stdin) - empty standard input


Object Heap
Unused Memory
Call Stack
Static vars
Bytecodes
OS / JVM
Base Pointer
Method n Activation Record
. . .
Method 3 Activation Record
Method 2 Activation Record
Method 1 Activation Record
Program Counter
Local vars
Parameters
Return Address (PC value)
Previous base pointer
Return value

Pointers, by-value vs by-reference

- To access the address of the variable use &
- If you know the address and want to access data, use *
- Addresses have 32-bits (=>old, up to 4GB RAM) or 64-bit (up to 16EB RAM)
- When argument is **passed by-value**, its value is copied to function's stack and all the changes happen on the copy. Once function ends its stack is deleted and program returned to the stack of the caller function. No change is visible, except if value was returned and saved into variable of caller function.
- When argument is **passed by-reference** (address) all the value changes happen at that address, so they are visible after function ends
- In C, **pass by-value is default**
- To pass by-reference, you need to use & in front of the variable name

By-value vs by-reference example

```
# include <stdio.h>
int main() {
        char name[50];                          // name is an address of the first char in
name
    int age;                                                    // age is value, not address
    int* ageP = malloc(sizeof(int));        // ageP is address which data is int
    scanf("%s", name);          // stores input starting from address name
    scanf("%d", age);                       // memory fault error, storage address
required
    scanf("%d", &age);                  // stores input in the address of age value
    scanf("%d", ageP);                  // stores input in the address ageP
    printf("%d", age);                          // prints value of age
        printf("%d", *ageP);            // prints value of age stored in address ageP
        printf("%d", ageP);             // prints int version of address
        printf("%p", ageP);             // prints address of ageP in hexadecimal
        return 0;
}
```

Bit Manipulation (1)

powers of 10			powers of 2	
kilo	10^3	1,000	2^{10}	1,024
mega	10^6	1,000,000	2^{20}	1,048,576
giga	10^9	1,000,000,000	2^{30}	1,073,741,824
tera	10^{12}	1,000,000,000,000	2^{40}	1,099,511,627,776
peta	10^{15}	1,000,000,000,000,000	2^{50}	1,125,899,906,842,624
exa	10^{18}	1,000,000,000,000,000,000	2^{60}	1,152,921,504,606,846,976
zetta	10^{21}	1,000,000,000,000,000,000,000	2^{70}	1,180,591,620,717,411,303,424
yotta	10^{24}	1,000,000,000,000,000,000,000,000	2^{80}	1,208,925,819,614,629,174,706,176

Decimal to binary: 140

Divide: 140/2 = 70; 70/2 = 35; 35/2 = 17; 17/2 = 8; 8/2 = 4; 4/2 = 2; 2/2 = 1; 1/2 = 0;

Modules: 140%2=**0**; 70%2=**0**; 35%2=**1**; 17%2=**1**; 8%2=**0**; 4%2=**0**; 2%2=**0**; 1%2=**1**;

Result (read modules from right to left): 1000 1100

Bit Manipulation (2)

Binary to decimal: 1000 1100

- Multiply each digit by 2^n, where n is position from the right, sum:
  - $\mathbf{1}*2^8 + \mathbf{0}*2^7 + \mathbf{0}*2^6 + \mathbf{0}*2^5 + \mathbf{1}*2^4 + \mathbf{1}*2^3 + \mathbf{0}*2^2 + \mathbf{0}*2^1$

Binary addition: do right alignment of numbers and add digits in same position.

- Sum numbers from right to left. Append smaller number with 0 on the left.
- If sum is 2 write 0 and add one to the position on the left.
- Example: 1000 1100 (decimal 140)
- + 0001 1001 (decimal 25)
- Result: 1010 0101 (decimal 165)

Bitwise Negation: ~1000 1100 => 0111 0011 (decimal 115), 0 goes to 1 and 1 to 0

Bitwise AND: do right alignment of numbers. Append smaller with 0 on the left.

- Rules: 0 & 0 => 0; 0 & 1 => 0; 1 & 0 => 0; 1 & 1 => 1
- Example: 1000 1100 (decimal 140)
- & 0001 1001 (decimal 25)
- Result: 0000 1000 (decimal 8, always <= than both operands)

Bit Manipulation (2)

Bitwise OR: do right alignment of numbers. Append smaller with 0 on the left.

- Rules: 0 | 0 => 0; 0 | 1 => 1; 1 | 0 => 1; 1 | 1 => 1
- Example: 1000 1100 (decimal 140)
- & 0001 1001 (decimal 25)
- Result: 1001 1101 (decimal 157, always >= than both operands)

Bitwise XOR: do right alignment of numbers. Append smaller with 0 on the left.

- Rules: 0 ^ 0 => 0; 0 ^ 1 => 1; 1 ^ 0 => 1; 1 ^ 1 => 0
- Example: 1000 1100 (decimal 140)
- & 0001 1001 (decimal 25)
- Result: 1001 0101 (decimal 149)

What is the difference?: ! vs ~; && vs &; || vs |

- Yellow operand is logical operator on the level of whole number
- Orange operand is bitwise operator and compares bit-by-bit in number

Bit Manipulation (3)

Shift right >>: shifts all bits to the right X times and X rightmost bits are deleted. Same as division by 2^X.

- 1000 1100 >> 2; gives 1000 11 which is decimal 35, (ie. 140 / 2^2 = 35)

Shift left <<: shifts all bits to the left X times and zeros are added to the X rightmost bits. Same as multiplication by 2^X.

- 1000 1100 << 3; gives 100 0110 0000, decimal 1120 (ie. 140*2^3=1120)

Binary to octal:

- 1000 1100 = 10 001 100 => 10 becomes 1 * 2^1 + 0 * 2^0, ie. **2**. 001 becomes 0 * 2^2 + 0 * 2^1 + 1 * 2^0, ie. **1**. 100 becomes 1 * 2^2 + 0 * 2^1 + 0 * 2^0, ie. **4**. So, **214**.

Hexadecimal to octal:

- 1000 1100 => 1000 becomes 1 * 2^3 + 0 * 2^2+ 0 * 2^1 + 0 * 2^0, ie. **8**. 1100 becomes 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0, ie. **C**. So, hexadecimal number is **8C.**

Octal to binary: same as decimal to binary, but divide and take moduo with 8

Hexadecimal to binary: same as above, but divide and take moduo with 16

Bit Manipulation (4)

Two's complement of X is ~X + 1:

- ~1000 1100 + 1 = 0111 0011 + 1 = 0111 0100

Binary subtraction:

- Example: 1000 1100 (decimal 140)
- \- 0001 1001 (decimal 25)
- Result: 0111 0011 (decimal 115)

Negative number is two's complement of its positive value (starts with 1):

- 25 is 0001 1001 => -25 is ~0001 1001 + 1 = 1110 0110 + 1 = 1110 0111

Float numbers to binary float numbers:

- 0.125 * 2 = **0**.25, 0.25 * 2 = **0**.5, 0.5 * 2 = **1**.0 => **0.001**
- 0.675*2=**1**.35, 0.35*2=**0**.7, 0.7*2=**1**.4, 0.4*2=**0**.8, 0.8*2=**1**.6,...=> 0.10101

INT_MIN and INT_MAX is the range of integers that can be stored.

- What happens when we do INT_MAX + 5? We get INT_MIN + 4
- What happens when we do INT_MIN - 10? We get INT_MAX - 9

Byte ordering

- **Big endian:** most significant byte first: as we are used to write numbers
- **Little endian**: least significant byte first: used in most computers to make bitwise operations easier

- int i=0x01234567

big endian

address	value
1000	01
1001	23
1002	45
1003	67

little endian

address	value
1000	67
1001	45
1002	23
1003	01

Operations with pointers

- Addition: pointer + integer increments value of pointer: int i, *p; p = p+i;
- Subtraction: pointer - pointer results in integer which says how distant two pointers are (eg. find position of element in array): int i, *p, *q; i = p - q;
- No other operations are possible

Check bitwise.c code
Check lab4.c code