C Programming Lecture 10

Thursday, 20 June 2024 8:09 PM

Example

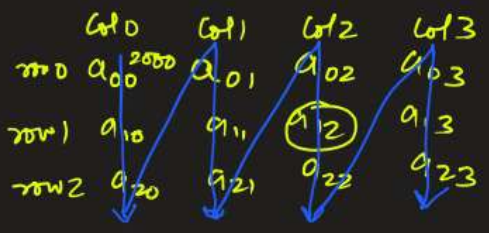
Consider a two-dimensional array A with 3 rows and 4 columns stored in memory in column-major order. Given the base address of the array A is 2000, and indices i = 1 (row index) and j = 2 (column index), what is the address of the element A[1][2]? (Assume the array stores integers which hold 4 B of memory)

Sol. A : 3 rows, 4 columns 3 × 4 matrix

add of $a[1][2] = 2000 + 2 \times 3 \times 4 + 1 \times 4$

$= 2000 + 24 + 4$

$= \boxed{2028}$



2028

Character Arrays and Strings

The string can be defined as the one-dimensional array of characters terminated by a null ('\0'). The character array or the string is used to manipulate text such as word or sentences.

Each character in the array occupies one byte of memory, and the last character must always be 0. The termination character ('\0') is important in a string since it is the only way to identify where the string ends.

When we define a string as char s[10], the character s[10] is implicitly initialized with the null in the memory.

```c
#include<stdio.h>
void main(){
    char s[] = "Hello";
    printf("%d", sizeof(s));
}
```

(6)

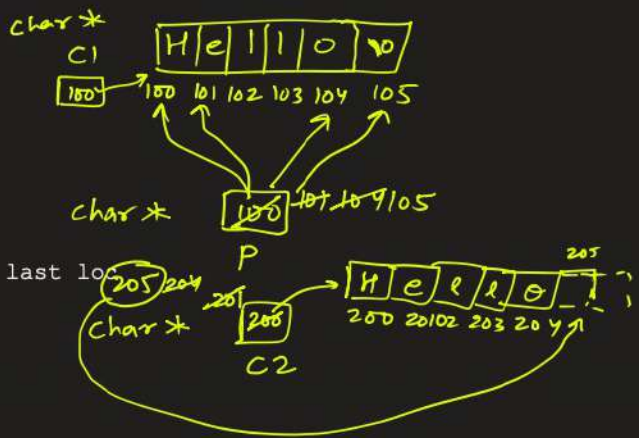
int a[] = {1,2,3};
char c[] = {'a', 'b', 'c'}; ✓
char c[] = "abc"; ✓

```c
#include<stdio.h>
void main(){
    char c1[] = "Hello";
    char c2[] = {'H', 'e', 'l', 'l', 'o'};
    char* c3 = "Hello";
    printf("%s \n", c1);
    printf("%s \n", c2);
    printf("%s \n", c3);
}
```

World Then at that moment it will also print Hello World

```c
#include<stdio.h>
void main(){
    char* c1 = "Hello";
    char* p = c1;
    char* c2;
    while(*p) {
        *c2++ = *p++;
    }
    printf("%s", c2); // Prints nothing as c2 is pointing to last lo
}
```

post > pre = *
right to left



*c2 = *p

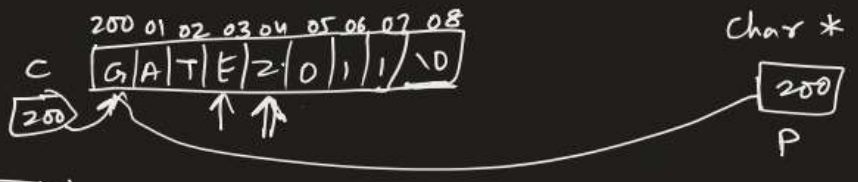
GATE CS 2011

What does the following fragment of C-program print?

```
char c[] = "GATE2011";
char *p = c;
printf ("%s", p + p[3] - p[1]);
```

A) GATE 2011 B) E 2011 C) 2011 D) 011

Sol.



$p[3] = *(p+3)$

$= *(203)$

$p[3] = 'E'$

$p[1] = *(201) = 'A'$

2011

p + 'E' - 'A';

p + 4 = 204

x+4 x

=x=

Functions in C

- A block of code with a unique name performing some specific task, which can be used multiple times throughout the execution
- The most important function in C is "main" function
- Main function is the entry point of every C program execution, because the compiler commands the thread to start execution from main function
- Using functions is a good practice as it provides features of modularity and reusability

	C function aspects	Syntax
1	Function declaration	`return_type function_name (argument list);`
2	Function call	`function_name (argument_list)`
3	Function definition	`return_type function_name (argument list) {function body;}`

Function Structure / Definition

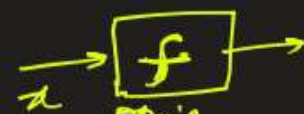
```
return_type function_name(parameter1_type p1, parameter2_type p2, ...) {
   // processing

   return x;
}
```

Math. functions

$f(x) = x^3 + 3x^2 + 2$

$y = f(x)$

$y = f(x_1, x_2, x_3)$




```
int f(x){
    opn's
    return (y)
}
```

```
int f(_x1, _x2, _x3){
    ~~~~~~~~~~~~~~~~~
    return y
}
```

Function Prototype (Declaration)

- Function prototype only mentions the interface of the function hiding the details of task to be done by the function
- Function prototype is declaration of the function which specifies the following:
  - Function name
  - Data types of the parameters
  - Return type of the function
- Function prototypes are useful for forward declarations (declaring prior to the definition)
- e.g.
  - `int function(int, char, float, int);`

Function Definition

Function definition consists of two parts :

- Function header : Same as function prototype, includes the parameters' variable names as well
- Function body : Code inside the curly braces written to perform that specific task the function is meant for
- e.g.

```
int function(int a, char b, float c, int d)   -> function header
{
return a+b+c+d;                               } function body
}
```

```
int add_integers(int a, int b);

add_integers(3,4);

int add_integers(int a, int b){
    return a+b;
}
```

Calling a function

- Function call is made when we require to perform the task the function is accomplishing
- While calling the function, we pass real values in accordance with the data type list of parameters of function prototype
- These real values which are passed at function calling are known as **arguments/actual parameters** to the function
- e.g.

```
int out = function(5, 'a', 1.167, 2021)
```

Formal parameters

```
int add (int a, int b) {
    int c = a+b;
    return c;
}
int k=5;
add ( 3, k );
```

actual parameters

Examples

- Write a function to find number of vowels in a string

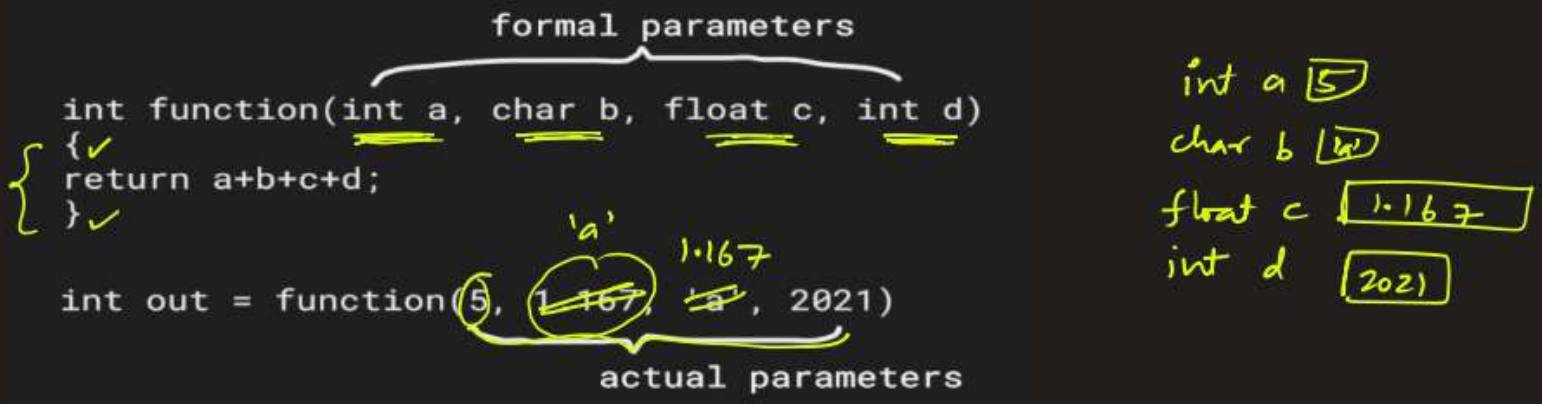
```
#include<stdio.h>
int is_vowel(charc){
    char vowels[] = "AEIOUaeiou";
    for(int i=0; i<sizeof(vowels)-1; i++) {
        if(c==vowels[i])
            return1;
    }
    return0;
}

int num_vowels(char str[], int size){
    int result = 0;
    for(int i=0; i<size; i++) {
        if(is_vowel(str[i]))
            result++;
    }
    return result;
}

void main(){
    char c1[] = "Hello World!";
    printf("%d", num_vowels(c1, sizeof(c1)-1));
}
```

[HW]

- Write a function to check if a string is a palindrome or not

Actual and formal parameters

Actual parameters : Arguments or values passed to the function while calling
Formal parameters : Variables used by the function to store received values to that function



Ways to call a function

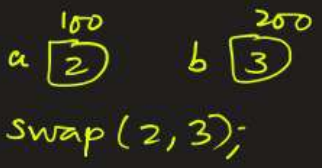
There are two ways of calling a function :

1. Call by Value
2. Call by Reference

Call by Value

- This method of calling **copies the values** from actual parameters to the formal parameters
- Both actual and formal parameters hence take separate memories of their own

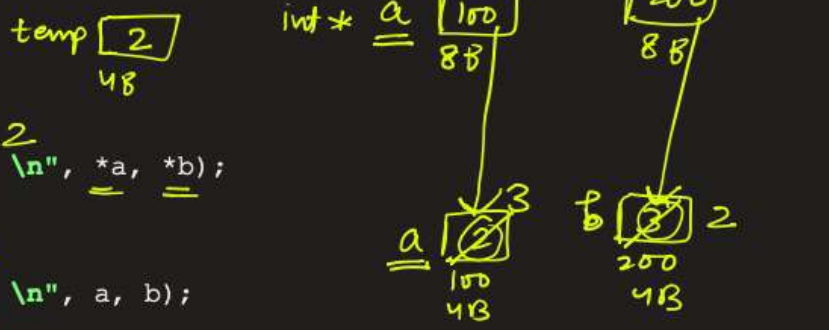
```c
#include<stdio.h>
void swap(int a, int b){
    int temp = a;
    a = b;
    b = temp;
    printf("Inside function: a = %d, b = %d \n", a, b);
}
int main(){
    int a = 2, b = 3;
    printf("Before swapping: a = %d, b = %d \n", a, b);
    swap(a, b);
    printf("After swapping: a = %d, b = %d \n", a, b);
    return 0;
}
```



Call by Reference

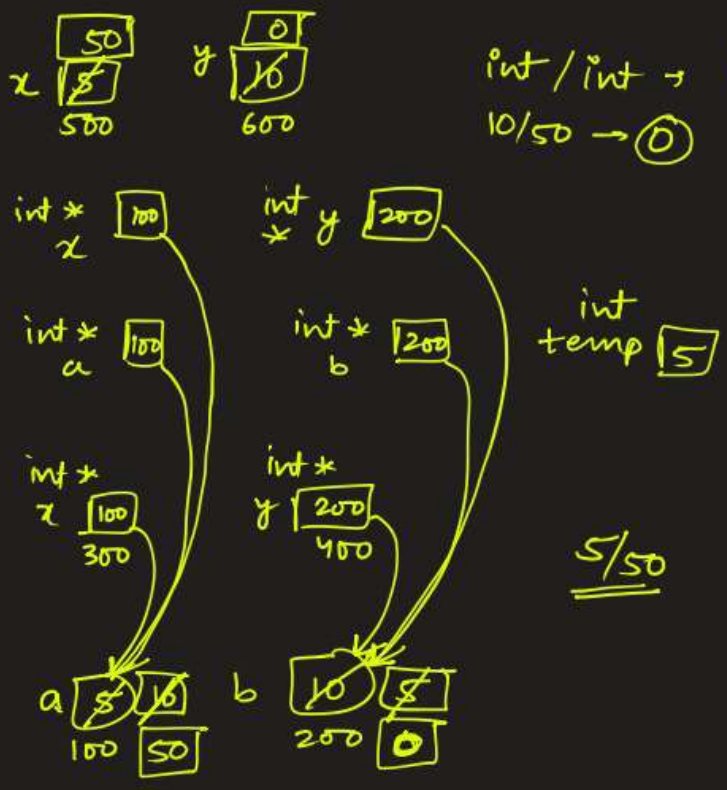
- In this method of calling instead of passing variable, address of the memory location that variable is passed
- In formal parameters, pointer is used to store the address
- Hence the changes are all made in the variable placed at that particular memory location only

```c
#include<stdio.h>
void swap(int* a, int* b){
    int temp = *a;
    *a = *b;
    *b = temp;
    printf("Inside function: a = %d, b = %d \n", *a, *b);
}
int main(){
    int a = 2, b = 3;
    printf("Before swapping: a = %d, b = %d \n", a, b);
    swap(&a, &b);
    printf("After swapping: a = %d, b = %d \n", a, b);
    return 0;
}
```



Example:

```c
#include<stdio.h>
void modifyValue(int x, int y){
    x = x * y;
    y = y / x;
}
void modifyReference(int* x, int* y){
    *x = *x * (*y);
    *y = *y / (*x);
}
void swap(int* a, int* b){
    int temp = *a;
    *a = *b;
    *b = temp;
}
void process(int* x, int* y){
    modifyValue(*x, *y);
    swap(x, y);
    modifyReference(x, y);
}
int main(){
    int a = 5, b = 10;
    process(&a, &b);
    printf("a = %d, b = %d\n", a, b);
    return 0;
}
```



What will be the output of the above program?

a) a = 5, b = 10

b) a = 10, b = 5

c) a = 50, b = 0

d) Compilation Error