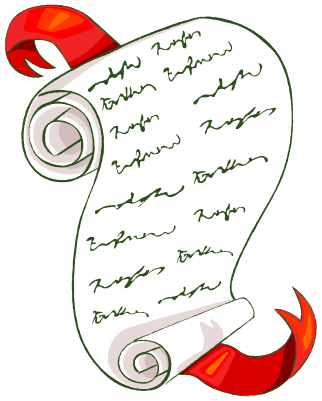
C-Refresher: Session 07
Pointers and Functions

Arif Butt
Summer 2017

I am Thankful to my student Muhammad Zubair bcsf14m029@pucit.edu.pk for preparation of these slides in accordance with my video lectures at

http://www.arifbutt.me/category/c-behind-the-curtain/

Today's Agenda



- Introduction to Pointers and Functions
- Concept of Activation Record
- Passing Arguments to Function by Pointers
- Passing Pointer to a Constant
- Passing 1D Integer Array to Function
- Passing 2D Integer Array to Function
- Passing Array of Character of Strings to Function
- Returning a Pointer from a Function

Pointers and Functions

/*let's start with this example function for swapping the values of two variables*/

```
#include<stdio.h>
void swap(int,int);
int main(){
  int n1=7, n2=10;
  printf("Before swap:\tn1=%d, n2=%d\n",n1,n2);
  swap(n1,n2); //calling swap() function
  printf("After swap:\tn1=%d, n2=%d\n",n1,n2);
  return 0;}
void swap(int num1,int num2){ //for swapping values
  int temp=num1;
  num1=num2;
  num2=temp;}
```

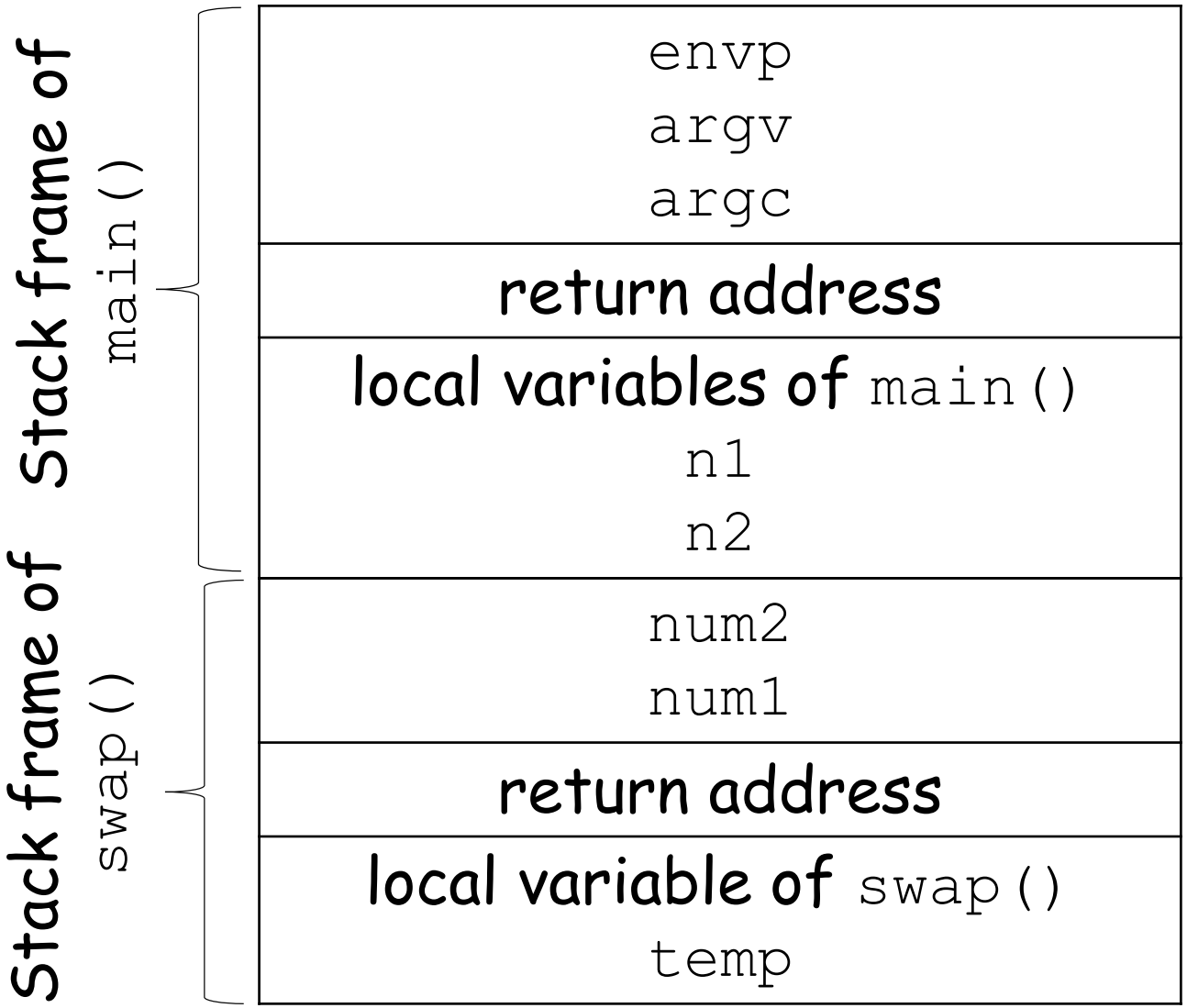
Pointers and Functions(cont…)

- Output of the above program is:

```
Before swap:      n1=7, n2=10
After swap:       n1=7, n2=10
```

- You see that the values have not been swapped
- Let's learn this by drawing the stack frame of this program
- **Note:** By stack frame, it means the activation record

Pointers and Functions(cont…)



arguments of `main()` in reverse order

return address is the address of the calling function of `main()`

return address is the address of the statement in `main()` where the `swap()` is going to fall back

Pointers and Functions(cont…)

- **Reason for no swap in above program**

- Two functions `main()` and `swap()` have different stack frames
- There is no relation between the variables of stack frames of two functions
- So a swap didn't take place
- **Solution Program:**
- Now let's write another program for swapping the values using **pointers**

Pointers and Functions(cont…)

```
#include<stdio.h>
void swap(int* const,int* const);
int main(){
  int n1=7,n2=10;
  printf("Before swap:\tn1=%d, n2=%d\n",n1,n2);
  swap(&n1,&n2);     //addresses are being passed
  printf("After swap:\tn1=%d, n2=%d\n",n1,n2);
  return 0;}
void swap(int* const pnum1,int* const pnum2){
  int temp=*pnum1;
  *pnum1=*pnum2;
  *pnum2=temp;}
  //pointers have swapped values at the addresses passed
```

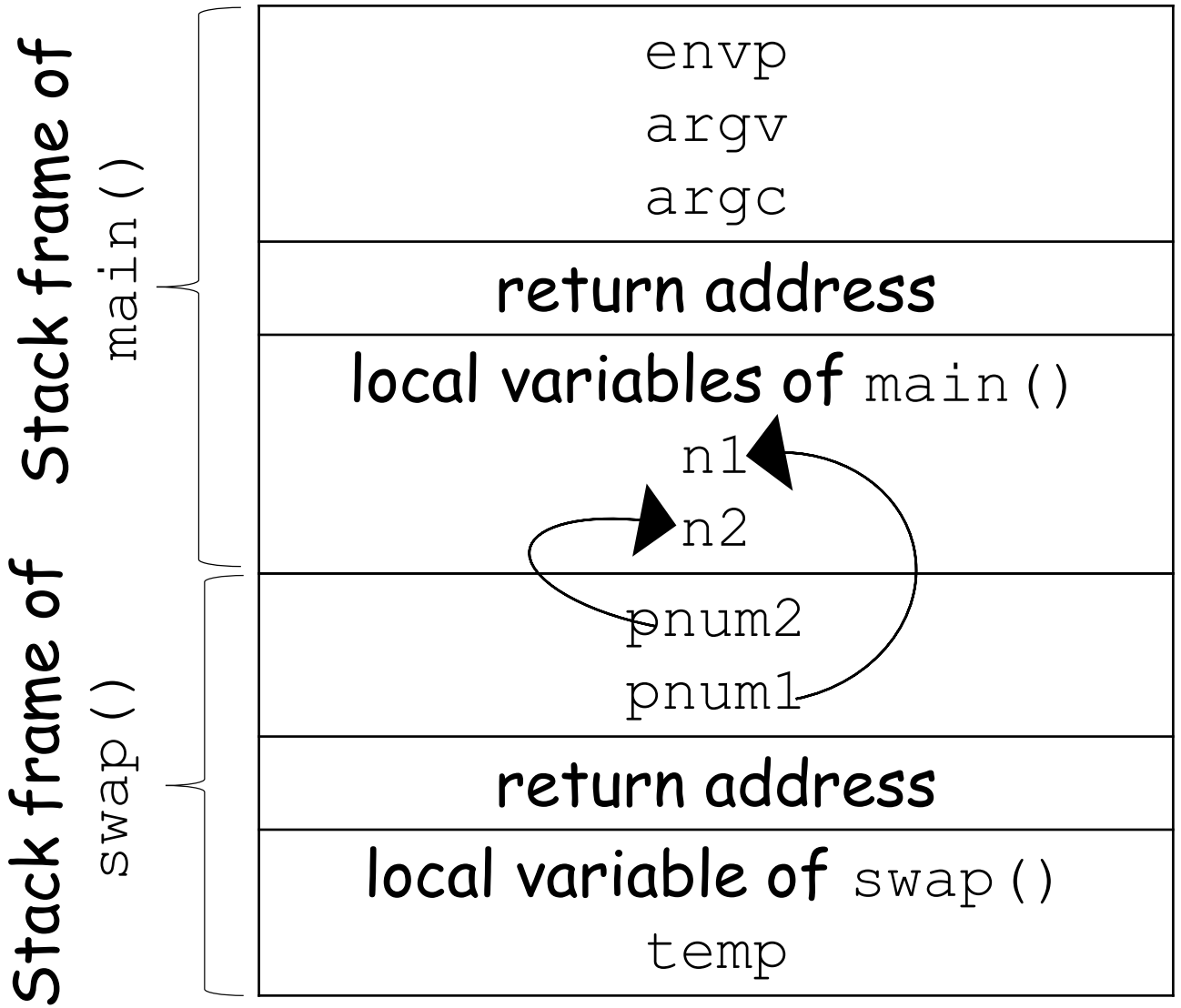
Pointers and Functions(cont…)

- Output of the above program is:

```
Before swap:      n1=7, n2=10
After swap:       n1=10, n2=7
```

- You see that a swap has occurred
- Now let's understand this by drawing the stack frame of this program

Pointers and Functions(cont…)



arguments of `main()` in reverse order

return address is the address of the calling function of `main()`

return address is the address of the statement in `main()` where the `swap()` is going to fall back

Pointers and Functions(cont…)

- **Reason for why a swap occurred**

- Although Two functions `main()` and `swap()` have different stack frames
- There is a relation between the variables of stack frames of two functions as the pointers `pnum1` and `pnum2` in the stack frame of `swap()` are pointing to the variables `n1` and `n2` in `main()`, respectively
- And the code swaps the values at the locations of `pnum1` and `pnum2` are pointing
- So a swap has occurred

Pointers and Functions(cont…)

❑Contents of Stack frame

i. Function arguments

ii. Return Address

iii. Local variables to functions

- Other than these there are two pointers that are used by the runtime system to manage the stack

iv. **Stack Pointer**, it points to the top of the stack. ESP is the register in intel for holding its value

v. **Base Pointer**, it points to the location within the frame where the return address is saved. In intel, register for holding its value is EBP

Pointers and Functions(cont…)

❑Primary Reason

- Primary reason for using pointers with functions is **"to allow the function to modify the data"** in the caller function

- **Note:** There is always one stack frame that is active and that is the top most stack frame

Passing Pointer to a Constant

- Let's start with a function example
- `void f1(const int* n1,int*n2){` /* `const int* n1` says that `n1` is a non-constant pointer pointing to a data that is constant*/

```
*n2=*n1;      //OK as data pointed by n2 is non-constant
*n1=*n2;      /*error as data pointed by n1 is constant so cannot be changed*/
(*n1)++;      /*error as data pointed by n1 cannot be incremented*/
*n1++;}       /*OK as n1 is not a constant so its value can be changed*/
```

Passing a 1D array to a Function

- We can pass a 1D array to a function using any of the following two ways

❑Option 1:(Using Array notation)

- `void print1(int arr[],const int size)`
  - //here the subscript indicates that an array will be passed to `arr`
  - //size is necessary to be passed as we do not know that how many elements are there inside the array

Passing a 1D array to a Function(cont…)

❑Option2:(Using Pointer notation)

- `void print2(int* ptr,const int size)`
  - `ptr` can be passed the address of any integer memory, we will pass it the address of the array so that we can use `ptr` as array pointer in the function

Passing a 1D array to a Function(cont…)

/*Program showing passing a 1D array to a function using array notation and pointer notation*/

```
#include<stdio.h>
void print1(int [],const int);//array notation
void print2(int*,const int);   //pointer notation
int main(){
  int arr[5]={1,2,3,4,5};
  printf("Printing 1D array using array notation:\n");
  print1(arr,5);/*arr holds the starting address of the
  array*/
  printf("Printing 1D array using pointer notation:\n");
  print2(arr,5);     //calling in the same way as print1()
  return 0;}
```

Passing a 1D array to a Function(cont…)

```
void print1(int arr[],const int size){
  for(int i=0;i<size;i++)
    printf("%d\t",*(arr+i));   /*We can also use
 subscript notation*/
  printf("\n");
}
void print2(int* arr,const int size){
  for(int i=0;i<size;i++)
    printf("%d\t",arr[i]);     /* We can also use pointer
 notation*/
  printf("\n");
}
```

Passing a 1D array to a Function(cont…)

- Output of the above program is:

```
Printing 1D array using array notation:
1  2    3    4    5
Printing 1D array using pointer notation:
1  2    3    4    5
```

Passing a 2D Array to a Function

- There are three ways for this
- e.g. `int arr[4][3]={1,2,3,…,12};` //array declared in `main()`

❑Option 1:(Using Array notation)

- `void print1(int arr[][3],const int rows);`
  - In array notation, the no. of columns are passed along with the array
  - And no. of rows are passed explicitly

❑Option 2:(Using Pointer notation)

- `void print2(int (*arr)[3],const int rows);`
  - The first argument says that `arr` is a pointer to a 1D array of integers of size 3

Passing a 2D array to a Function(cont…)

❑Option 3:(Use a simple pointer)

- `void print3(int* arr, const int rows, const int cols){`
  - //here first argument is a pointer to an integer, and in this pointer address of the first element of the array is passed as an argument
  - //Then there are no. of rows and cols

```
for(int i=0;i<rows;i++){
   for(int j=0;j<cols;j++)
      printf("%d\t",*(arr+(i*cols)+j));    /*cannot
      use subscript notation here*/
   printf("\n");
}}
```

Passing a 2D array to a Function(cont…)

- `*(arr+(i*cols)+j)`

- In this statement, `i*cols` steps a complete row for every iteration of the outer loop
- And then `+j` is used to iterate through the elements of the row
- Outer `*` is used to dereference the value of the pointer `arr`

Passing a 2D array to a Function(cont…)

/*Program showing passing a 2D array to a function using array notation, pointer notation and simple pointer to an `int` notation*/

```
#include<stdio.h>
void print1(int [][3],const int);
void print2(int (*)[3],const int);
void print3(int*,const int,const int);
int main(){
//int arr[4][3]={1,2,3,4,5,6,7,8,9,10,11,12};
  int arr[][3]={1,2,3,4,5,6,7,8,9,10,11,12};
  printf("Passing 2D array using array notation:\n");
  print1(arr,4);
```

Passing a 2D array to a Function(cont…)

```
  printf("Passing 2D array using pointer to an
  array:\n");
  print2(arr,4);
  printf("Passing 2D array using pointer to int:\n");
  print3(&arr[0][0],4,3);
  return 0;}
void print1(int arr[][3],const int rows){
  for(int i=0;i<rows;i++){
    for(int j=0;j<3;j++)
      printf("%d\t",arr[i][j]);
    printf("\n");
  }
}
```

Passing a 2D array to a Function(cont…)

```
void print2(int (*arr)[3],const int rows){
  for(int i=0;i<rows;i++){
    for(int j=0;j<3;j++)
      printf("%d\t",arr[i][j]);
    printf("\n");
  }}
void print3(int* arr,const int rows,const int cols){
  for(int i=0;i<rows;i++){
    for(int j=0;j<cols;j++)
      printf("%d\t",*(arr+(i*cols)+j)); /*cannot use
double subscript operator here*/
    printf("\n");
  }
}
```

Passing a 2D array to a Function(cont…)

- Output of the above program is:

```
Passing 2D array using array notation:
1  2    3
4  5    6
7  8    9
10 11   12
Passing 2D array using pointer to an array:
1  2    3
4  5    6
7  8    9
10 11   12
Passing 2D array using pointer to int:
1  2    3
4  5    6
7  8    9
10 11   12
```

Passing Array of Character of Strings to Function

- `void print(char* names[],int count){`
  - //first argument is an array of character of strings
  - //second argument is the size of the array
- To call the function, we will pass the array name and its size

Passing Array of Character of Strings to Function(cont…)

```
#include<stdio.h>
void print(char*[],int);
int main(){
char* names[5]={"ArifButt","Rauf","Maaz","Hadeed","Mujahid"};
  print(names,5);
  return 0;
}
void print(char* names[],int count){
  for(int i=0;i<count;i++)
    printf("%s\n",names[i]);
}
```

Passing Array of Character of Strings to Function(cont…)

- Output of the above program is:

```
Arif Butt
Rauf
Maaz
Hadeed
Mujahid
```

Returning a Pointer from a Function

/*Let's explain this through program examples*/

//the program computes the square of a number

```
#include<stdio.h>
int* square(int);
int main(){
  int a=5;
  int* result = square(a);
  printf("Square of %d is %d\n",a,*result);
  return 0;}
int* square(int n){
  int result=n*n;
  return &result;}
```

/*returning the address of the local variable result*/

Returning a Pointer from a Function(cont…)

- Output of the above program is:

```
Segmentation fault (core dumped)
```
//an error!

- In this program, the address of a local variable 'result' is being returned to the `main()` from the `square()`
- As `result` is declared in the stack frame of `square()`, and when the stack frame of `square()` is popped out of the stack at the end of the function, `result` no more exists there. While we are returning its address to the `main()`,which causes segmentation fault when we try to access `result` in `main()`

Returning a Pointer from a Function(cont…)

```
/*A static variable is returned from the function*/
#include<stdio.h>
int* square(int);
int main(){
  int a=5;
  int* result = square(a);
  printf("Square of %d is %d\n",a,*result);
  return 0;}
int* square(int n){
  static int result;      //declared static
  result=n*n;
  return &result;}  //returning the value of static variable
```

Returning a Pointer from a Function(cont…)

- Output of the above program is:

```
Square of 5 is 25
```

- We have used here `static` keyword with the `result` variable, which causes `result` to retain its value between various calls to the function

❑Limitation in the above program

- The condition in the above program works OK for single threaded program, but can cause problems like race condition when this program is called by multiple threads
- Using global variable for returning values from function will also suffer from the same limitation

Returning a Pointer from a Function(cont…)

/*Now Option 1 is to allocate memory on heap in `square()` using `malloc` or `new` and then returning the address of that memory*/

```
#include<stdlib.h>
#include<stdio.h>
int* square(int);
int main(){
  int a=5;
  int* result = square(a);
  printf("Square of %d is %d\n",a,*result);
  free(result);  //freeing the memory on heap
  return 0;}
int* square(int n){
  int* result=(int*)malloc(sizeof(int)*1); //allocating memory
  *result=n*n;
  return result;}
```

Returning a Pointer from a Function(cont…)

- Output of the above program is:

```
Square of 5 is 25
```

- The memory allocated in the called function must be freed by the caller function, if we don't do this then it may cause heap leakage issue

Returning a Pointer from a Function(cont…)

```
/*Option 2: variable is declared in main() and that variable is passed as a pointer to square(), and the data of that variable is updated in square()*/
#include<stdlib.h>
#include<stdio.h>
void square(int,int*); /*the function may of may not return any value*/
int main(){
  int a=5;
  int result;
  square(a,&result);//address of local variable is being passed
  printf("Square of %d is %d\n",a,result);
  return 0;}
void square(int n,int* result){
  *result=n*n;}  /*square is stored at the address pointed by result*/
```

Returning a Pointer from a Function(cont…)

- Output of the above program is:

```
Square of 5 is 25
```

- So, these are the two ways of returning value from a function, you may use any of them depending upon your conditions

SUMMARY