Introduction to Programming

Week – *8*, Lecture – *1*

Functions in C – Part 2

SAURABH SRIVASTAVA

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

IIT KANPUR

What can we do with functions?

Part of a library, or defined by you, we have used functions for many aspects till now, e.g.

- Printing values on screen (using the good old `printf()` function)
- Calculating complex mathematical functions like square root (through the `sqrt()` function)
- Removing leading and trailing spaces from a string (with the `trim()` function we wrote) etc.

What can we do with functions?

Part of a library, or defined by you, we have used functions for many aspects till now, e.g.

- Printing values on screen (using the good old `printf()` function)
- Calculating complex mathematical functions like square root (through the `sqrt()` function)
- Removing leading and trailing spaces from a string (with the `trim()` function we wrote) etc.

However, there are some simple use cases, which can not be achieved using functions, e.g.

- Returning two values from the called function (we can return *one* array though…)

What can we do with functions?

Part of a library, or defined by you, we have used functions for many aspects till now, e.g.

- Printing values on screen (using the good old `printf()` function)
- Calculating complex mathematical functions like square root (through the `sqrt()` function)
- Removing leading and trailing spaces from a string (with the `trim()` function we wrote) etc.

However, there are some simple use cases, which can not be achieved using functions, e.g.

- Returning two values from the called function (we can return *one* array though…)

At times, you may need a function to return two or more values

- For such cases, we can use a trick that many programming languages provide

What can we do with functions?

Part of a library, or defined by you, we have used functions for many aspects till now, e.g.

- Printing values on screen (using the good old `printf()` function)
- Calculating complex mathematical functions like square root (through the `sqrt()` function)
- Removing leading and trailing spaces from a string (with the `trim()` function we wrote) etc.

However, there are some simple use cases, which can not be achieved using functions, e.g.

- Returning two values from the called function (we can return *one* array though…)

At times, you may need a function to return two or more values

- For such cases, we can use a trick that many programming languages provide

The trick is to pass parameters *by reference* and not *by value*

What is meant by *Pass by Value*?

In C, whenever a function call is made, parameters are passed *by value*

What is meant by *Pass by Value*?

In C, whenever a function call is made, parameters are passed *by value*

This means that the value of the actual parameters “travel” to the called function…

- … even if the passed parameter is a variable, all that matters, is its “current” value

What is meant by *Pass by Value*?

In C, whenever a function call is made, parameters are passed *by value*

This means that the value of the actual parameters “travel” to the called function…

- … even if the passed parameter is a variable, all that matters, is its “current” value

These values are copied to “new” memory locations…

- … allocated for the variables that act as formal parameters in the called function

Passing parameters by Value

```
function f()
{
    ...
    a = 10;
    g(a, 5);
    ...
}


function g(int a, int b)
{
    ...
    ...
    ...
}
```

For example, if `f()` is the calling function and `g()` is the called function

Passing parameters by Value

```
function f()
{
    ...
    a = 10;
    g(a, 5);
    ...
}


function g(int a, int b)
{
    ...
    ...
    ...
}
```

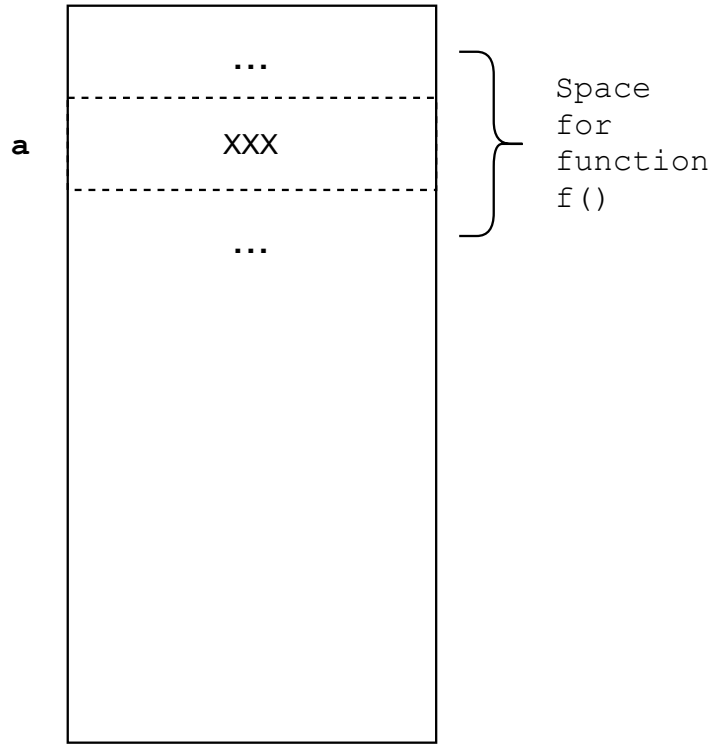
For example, if `f()` is the calling function and `g()` is the called function

Also, `g()` expects two parameters, `a` and `b` (both integers)

Passing parameters by Value

```
function f()
{
     ...
     a = 10;
     g(a, 5);
     ...
}


function g(int a, int b)
{
     ...
     ...
     ...
}
```

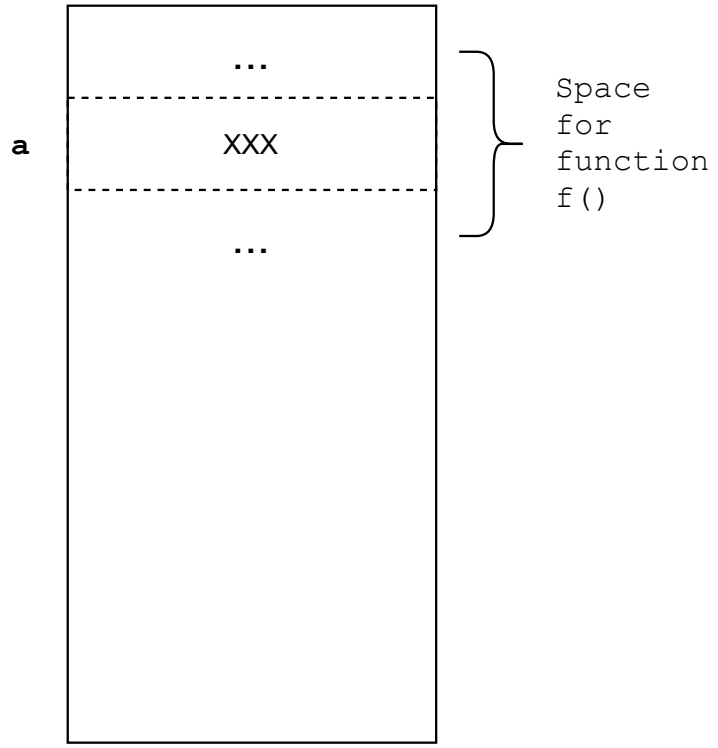


Some space in the memory is allocated for execution of `f()`

Passing parameters by Value

```
function f()
{
    ...
    a = 10;
    g(a, 5);
    ...
}


function g(int a, int b)
{
    ...
    ...
    ...
}
```



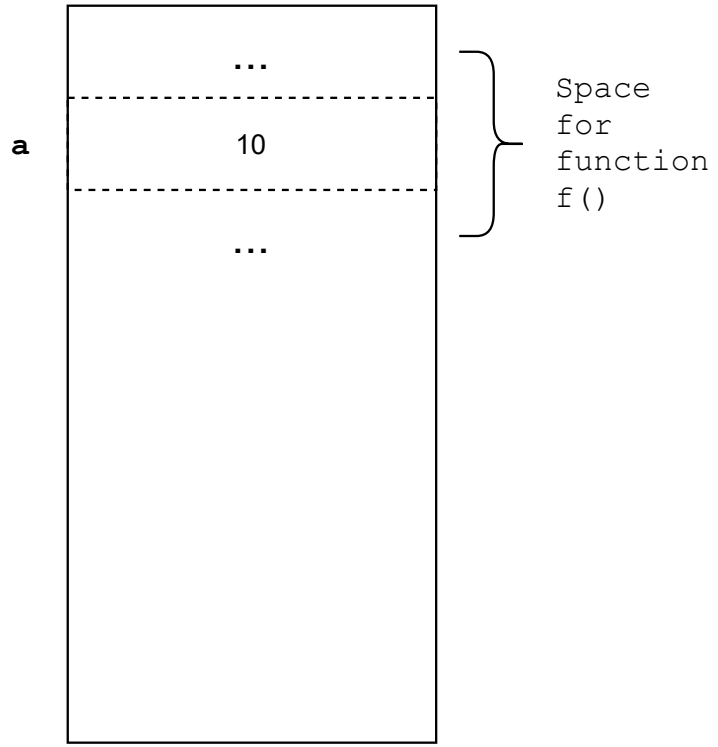
Some space in the memory is allocated for execution of `f()`

The local variable, `a`, is allocated some space in this block

Passing parameters by Value

```
function f()
{
    ...
    a = 10;
    g(a, 5);
    ...
}


function g(int a, int b)
{
    ...
    ...
    ...
}
```

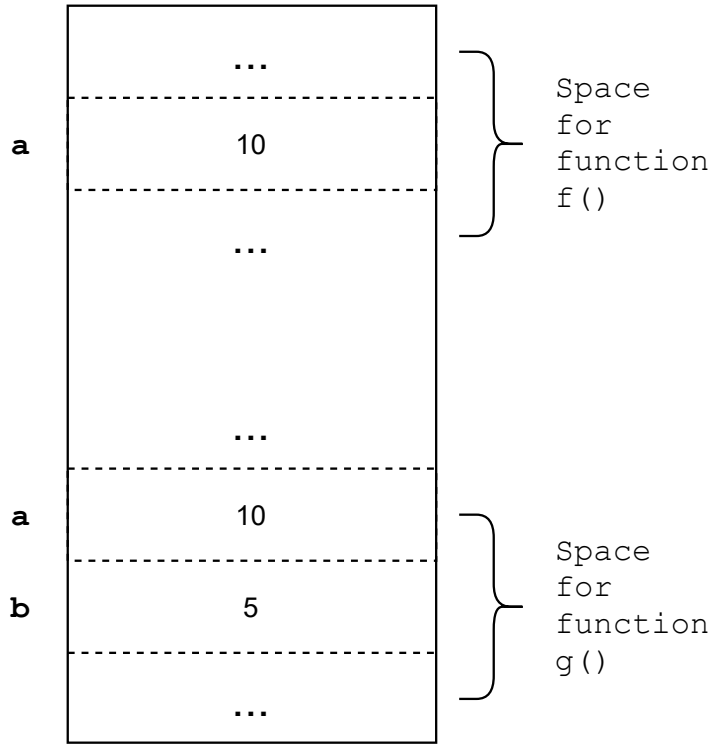


Assume that the ”current” value of `a` is set to `10`

Passing parameters by Value

```
function f()
{
    ...
    a = 10;
    g(a, 5);
    ...
}

function g(int a, int b)
{
    ...
    ...
    ...
}
```

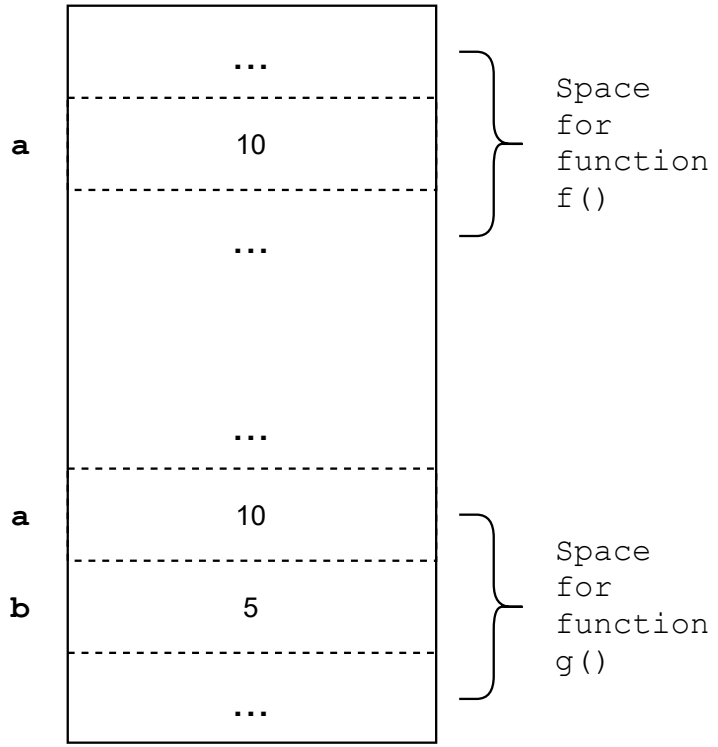


When `g()` is called, some more space is set aside for its execution

Passing parameters by Value

```
function f()
{
    ...
    a = 10;
    g(a, 5);
    ...
}

function g(int a, int b)
{
    ...
    ...
    ...
}
```



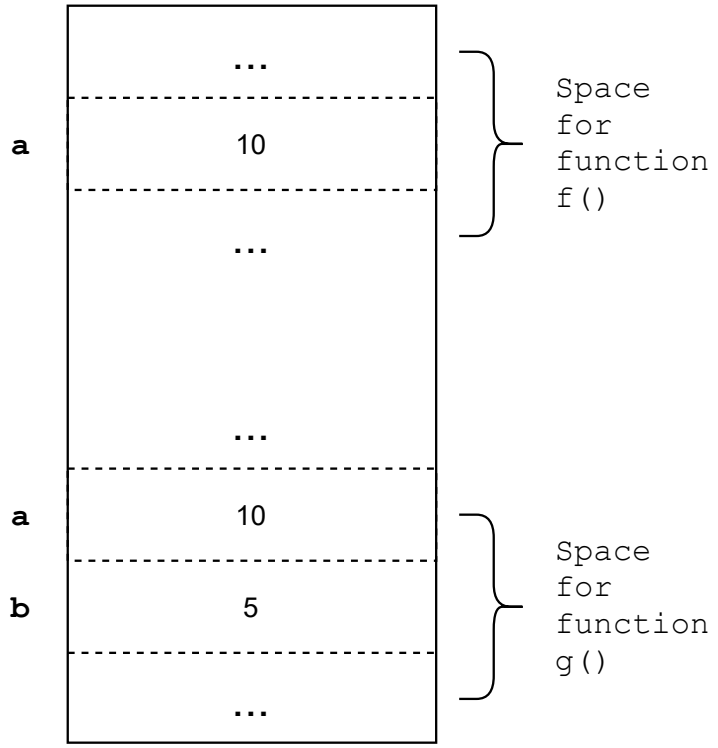
When `g()` is called, some more space is set aside for its execution

Two local variables, `a` and `b` are allocated some space in this block…

Passing parameters by Value

```
function f()
{
    ...
    a = 10;
    g(a, 5);
    ...
}

function g(int a, int b)
{
    ...
    ...
    ...
}
```



When `g()` is called, some more space is set aside for its execution

Two local variables, `a` and `b` are allocated some space in this block…

… and, they are initialised by copying values of actual parameters (i.e. `10` and `5`)

What is meant by *Pass by Value*?

In C, whenever a function call is made, parameters are passed *by value*

This means that the value of the actual parameters “travel” to the called function…

- … even if the passed parameter is a variable, all that matters, is its “current” value

These values are copied to “new” memory locations…

- … allocated for the variables that act as formal parameters in the called function

These memory locations are “active”, till the function executes, and are “freed” when it returns

What is meant by *Pass by Value*?

In C, whenever a function call is made, parameters are passed *by value*

This means that the value of the actual parameters “travel” to the called function…

- … even if the passed parameter is a variable, all that matters, is its “current” value

These values are copied to “new” memory locations…

- … allocated for the variables that act as formal parameters in the called function

These memory locations are “active”, till the function executes, and are “freed” when it returns

Since only the values travelled to the called function…

- … the memory allocated for (and hence, any variables in,) the calling function remain unaffected

What is meant by *Pass by Reference*?

The advantage of passing parameters by value, is that there are no “side effects” of a function call

- Here, side effects mean undesired changes in values of variables in the calling function, after the call

What is meant by *Pass by Reference*?

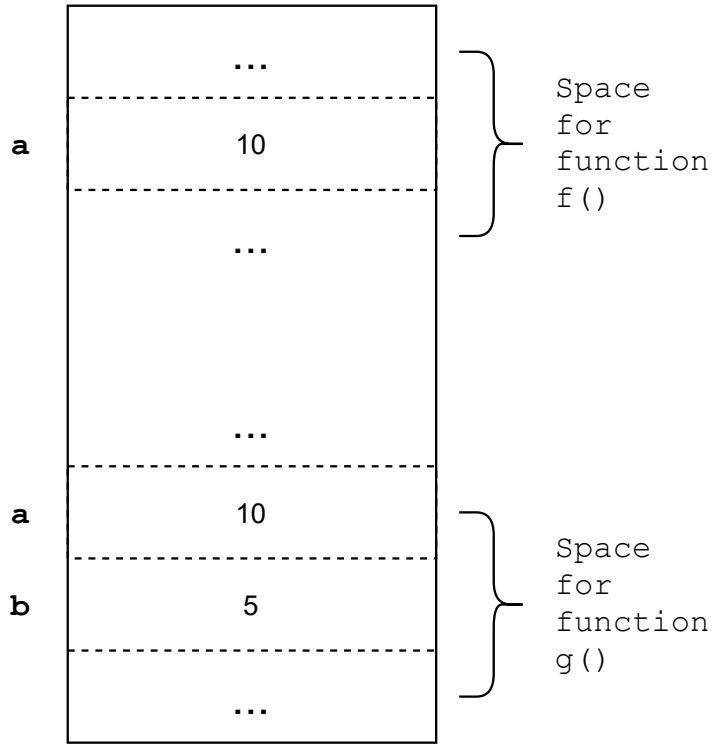
The advantage of passing parameters by value, is that there are no “side effects” of a function call

- Here, side effects mean undesired changes in values of variables in the calling function, after the call
- e.g. in the sample scenario, the value of `a` in `f()` is not affected by any changes to `a` in `g()`

Passing parameters by Value

```
function f()
{
     ...
     a = 10;
     g(a, 5);
     ...
}


function g(int a, int b)
{
     ...
     ...
     ...
}
```

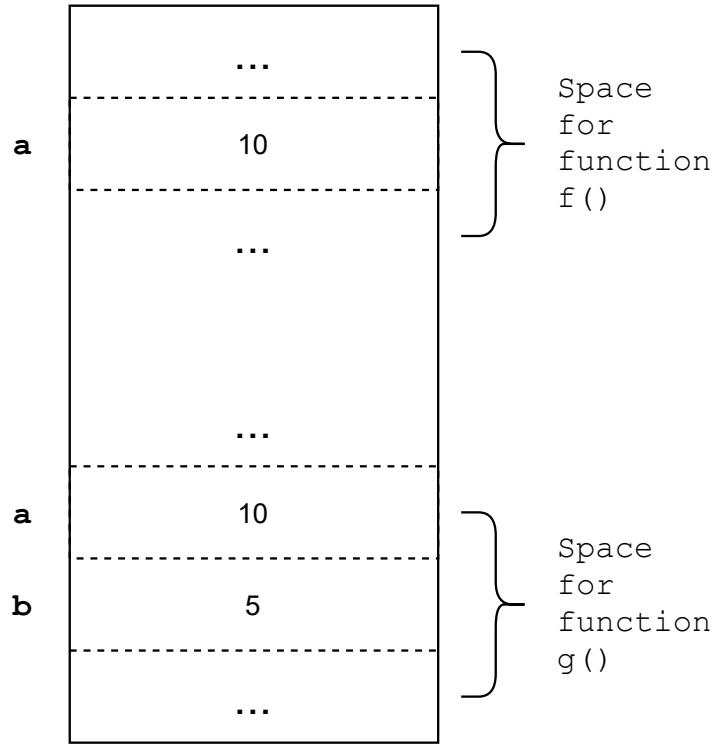


The two a variables here are at different memory locations

Passing parameters by Value

```
function f()
{
     ...
     a = 10;
     g(a, 5);
     ...
}


function g(int a, int b)
{
     ...
     ...
     ...
}
```



The two `a` variables here are at different memory locations

Thus, a change to `a` in `g()`, does not affect `a` in `f()`

What is meant by *Pass by Reference*?

The advantage of passing parameters by value, is that there are no “side effects” of a function call

- Here, side effects mean undesired changes in values of variables in the calling function, after the call
- e.g. in the sample scenario, the value of `a` in `f()` is not affected by any changes to `a` in `g()`

However, sometimes, we may need these side effects

- The most common example, is when we need more than one value to be returned from a function

What is meant by *Pass by Reference*?

The advantage of passing parameters by value, is that there are no “side effects” of a function call

- Here, side effects mean undesired changes in values of variables in the calling function, after the call
- e.g. in the sample scenario, the value of `a` in `f()` is not affected by any changes to `a` in `g()`

However, sometimes, we may need these side effects

- The most common example, is when we need more than one value to be returned from a function

In this case, a common trick is to *pass one or more parameters* to the function by *reference*

What is meant by *Pass by Reference*?

The advantage of passing parameters by value, is that there are no “side effects” of a function call

- Here, side effects mean undesired changes in values of variables in the calling function, after the call
- e.g. in the sample scenario, the value of `a` in `f()` is not affected by any changes to `a` in `g()`

However, sometimes, we may need these side effects

- The most common example, is when we need more than one value to be returned from a function

In this case, a common trick is to *pass one or more parameters* to the function by *reference*

This involves sending the address of a variable as a parameter value from the calling function…

- … and in the called function, using a pointer variable to store the address

Passing parameters by Reference

```
function f()
{
    ...
    a = 10;
    g(&a, 5);
    ...
}


function g(int *p, int b)
{
    ...
    ...
    ...
}
```

For example, in the previously discussed scenario, we can pass the first parameter by reference as well (the second parameter, here is a constant, so pass by reference is not applicable)

Passing parameters by Reference

```
function f()
{
    ...
    a = 10;
    g(&a, 5);
    ...
}


function g(int *p, int b)
{
    ...
    ...
    ...
}
```

For example, in the previously discussed scenario, we can pass the first parameter by reference as well (the second parameter, here is a constant, so pass by reference is not applicable)

We pass the address of `a` from `f()` and use a pointer variable `p` to store it in `g()`

What is meant by *Pass by Reference*?

The advantage of passing parameters by value, is that there are no “side effects” of a function call

- Here, side effects mean undesired changes in values of variables in the calling function, after the call
- e.g. in the sample scenario, the value of `a` in `f()` is not affected by any changes to `a` in `g()`

However, sometimes, we may need these side effects

- The most common example, is when we need more than one value to be returned from a function

In this case, a common trick is to *pass one or more parameters* to the function by *reference*

This involves sending the address of a variable as a parameter value from the calling function…

- … and in the called function, using a pointer variable to store the address

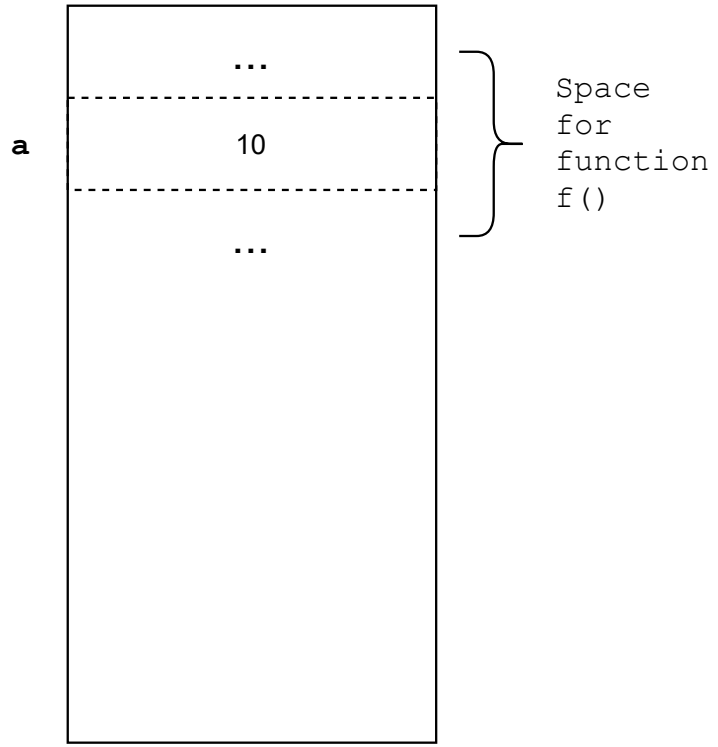
In the called function, we make changes using the address of the passed variable…

- … meaning that when the call finishes, the changes made in the function are visible in the calling function

Passing parameters by Reference

```
function f()
{
    ...
    a = 10;
    g(&a, 5);
    ...
}


function g(int *p, int b)
{
    ...
    ...
    ...
}
```

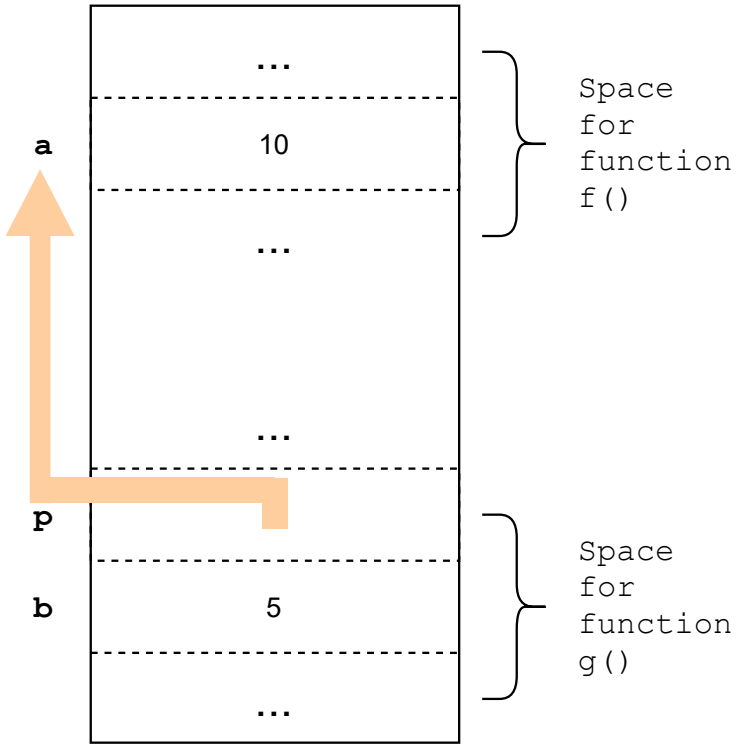


It doesn't matter now, what the current value of `a` is…

Passing parameters by Reference

```
function f()
{
    ...
    a = 10;
    g(&a, 5);
    ...
}

function g(int *p, int b)
{
    ...
    ...
    ...
}
```

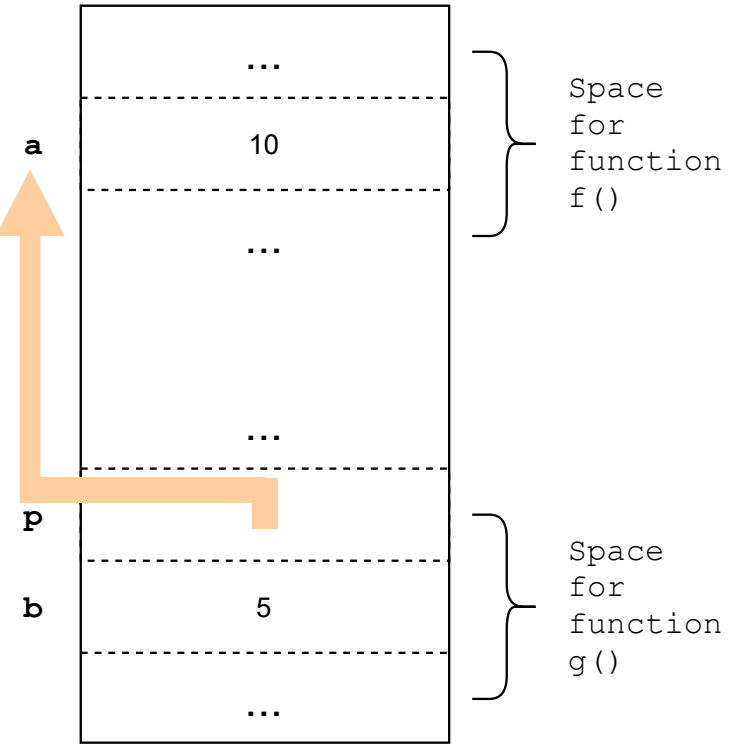


… because the memory location of `a`, is directly accessible to `g()`

Passing parameters by Reference

```
function f()
{
    ...
    a = 10;
    g(&a, 5);
    ...
}

function g(int *p, int b)
{
    ...
    ...
    ...
}
```



… because the memory location of `a`, is directly accessible to `g()`

Any changes made via this address (by using `*p`) happen directly at the location of `a`, making it visible to `f()` even after `g()` returns

What is meant by *Pass by Reference*?

The advantage of passing parameters by value, is that there are no “side effects” of a function call

- Here, side effects mean undesired changes in values of variables in the calling function, after the call
- e.g. in the sample scenario, the value of `a` in `f()` is not affected by any changes to `a` in `g()`

However, sometimes, we may need these side effects

- The most common example, is when we need more than one value to be returned from a function

In this case, a common trick is to *pass one or more parameters* to the function by *reference*

This involves sending the address of a variable as a parameter value from the calling function…

- … and in the called function, using a pointer variable to store the address

In the called function, we make changes using the address of the passed variable…

- … meaning that when the call finishes, the changes made in the function are visible in the calling function

This trick, thus, can be used to get *implicit* outputs from a function, even without returning them

Example – The *im*perfect swapping

```c
#include<stdio.h>

void imperfect_swap(int a, int b)
{
        int temp;
        printf("In function imperfect_swap()\n");
        printf("Before swap...\n");
        printf("a = %d, b = %d\n", a, b);
        printf("Swapping...\n");
        temp = a;
        a = b;
        b = temp;
        printf("After swap...\n");
        printf("a = %d, b = %d\n", a, b);
}

int main()
{
        int a = 5, b =15;
        printf("In function main()\n");
        printf("Before swap...\n");
        printf("a = %d, b = %d\n", a, b);
        printf("Calling the function to swap values...\n");
        imperfect_swap(a, b);
        printf("Call to the function returned\n");
        printf("a = %d, b = %d\n", a, b);
        return 0;
}
```

To see its relevance, let us see this “incorrect” attempt to use a function to swap values of two integer variables

Example – The *im*perfect swapping

```c
#include<stdio.h>

void imperfect_swap(int a, int b)
{
        int temp;
        printf("In function imperfect_swap()\n");
        printf("Before swap...\n");
        printf("a = %d, b = %d\n", a, b);
        printf("Swapping...\n");
        temp = a;
        a = b;
        b = temp;
        printf("After swap...\n");
        printf("a = %d, b = %d\n", a, b);
}

int main()
{
        int a = 5, b =15;
        printf("In function main()\n");
        printf("Before swap...\n");
        printf("a = %d, b = %d\n", a, b);
        printf("Calling the function to swap values...\n");
        imperfect_swap(a, b);
        printf("Call to the function returned\n");
        printf("a = %d, b = %d\n", a, b);
        return 0;
}
```

To see its relevance, let us see this “incorrect” attempt to use a function to swap values of two integer variables

In the first attempt, we are trying to do so, by passing the parameters by value

Example – The *im*perfect swapping

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 8$ gcc ImperfectSwap.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 8$ ./a.out
In function main()
Before swap...
a = 5, b = 15
Calling the function to swap values...
In function imperfect_swap()
Before swap...
a = 5, b = 15
Swapping...
After swap...
a = 15, b = 5
Call to the function returned
a = 5, b = 15
saurabh@saurabh-VirtualBox:~/C/examples/Week 8$
```

Example – The *im*perfect swapping

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 8$ gcc ImperfectSwap.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 8$ ./a.out
In function main()
Before swap...
a = 5, b = 15
Calling the function to swap values...
In function imperfect_swap()
Before swap...
a = 5, b = 15
Swapping...
After swap...
a = 15, b = 5
Call to the function returned
a = 5, b = 15
saurabh@saurabh-VirtualBox:~/C/examples/Week 8$
```

While the values change in the called function…

Example – The *im*perfect swapping

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 8$ gcc ImperfectSwap.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 8$ ./a.out
In function main()
Before swap...
a = 5, b = 15
Calling the function to swap values...
In function imperfect_swap()
Before swap...
a = 5, b = 15
Swapping...
After swap...
a = 15, b = 5
Call to the function returned
a = 5, b = 15
saurabh@saurabh-VirtualBox:~/C/examples/Week 8$
```

While the values change in the called function…

... they remain the same in the calling function

Example – The perfect swapping

```c
#include<stdio.h>

void perfect_swap(int *a, int *b)
{
        int temp;
        printf("In function perfect_swap()\n");
        printf("Before swap...\n");
        printf("a = %d, b = %d\n", *a, *b);
        printf("Swapping...\n");
        temp = *a;
        *a = *b;
        *b = temp;
        printf("After swap...\n");
        printf("a = %d, b = %d\n", *a, *b);
}

int main()
{
        int a = 5, b =15;
        printf("In function main()\n");
        printf("Before swap...\n");
        printf("a = %d, b = %d\n", a, b);
        printf("Calling the function to swap values...\n");
        perfect_swap(&a, &b);
        printf("Call to the function returned\n");
        printf("a = %d, b = %d\n", a, b);
        return 0;
}
```

In the second attempt, we change the type of parameter passing to reference

Example – The perfect swapping

```c
#include<stdio.h>

void perfect_swap(int *a, int *b)
{
        int temp;
        printf("In function perfect_swap()\n");
        printf("Before swap...\n");
        printf("a = %d, b = %d\n", *a, *b);
        printf("Swapping...\n");
        temp = *a;
        *a = *b;
        *b = temp;
        printf("After swap...\n");
        printf("a = %d, b = %d\n", *a, *b);
}

int main()
{
        int a = 5, b =15;
        printf("In function main()\n");
        printf("Before swap...\n");
        printf("a = %d, b = %d\n", a, b);
        printf("Calling the function to swap values...\n");
        perfect_swap(&a, &b);
        printf("Call to the function returned\n");
        printf("a = %d, b = %d\n", a, b);
        return 0;
}
```

In the second attempt, we change the type of parameter passing to reference

So, we now use two pointer variables to perform the swapping

Example – The perfect swapping

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 8$ gcc PerfectSwap.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 8$ ./a.out
In function main()
Before swap...
a = 5, b = 15
Calling the function to swap values...
In function perfect_swap()
Before swap...
a = 5, b = 15
Swapping...
After swap...
a = 15, b = 5
Call to the function returned
a = 15, b = 5
saurabh@saurabh-VirtualBox:~/C/examples/Week 8$
```

Example – The perfect swapping

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 8$ gcc PerfectSwap.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 8$ ./a.out
In function main()
Before swap...
a = 5, b = 15
Calling the function to swap values...
In function perfect_swap()
Before swap...
a = 5, b = 15
Swapping...
After swap...
a = 15, b = 5
Call to the function returned
a = 15, b = 5
saurabh@saurabh-VirtualBox:~/C/examples/Week 8$
```

Now, the changes made in the called function…

Example – The perfect swapping

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 8$ gcc PerfectSwap.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 8$ ./a.out
In function main()
Before swap...
a = 5, b = 15
Calling the function to swap values...
In function perfect_swap()
Before swap...
a = 5, b = 15
Swapping...
After swap...
a = 15, b = 5
Call to the function returned
a = 15, b = 5
saurabh@saurabh-VirtualBox:~/C/examples/Week 8$
```

Now, the changes made in the called function…

... are also visible in the calling function

Some additional points

We can use the same concept while returning values as well…

- … i.e., instead of returning a value, we can also return an address

Some additional points

We can use the same concept while returning values as well…

- … i.e., instead of returning a value, we can also return an address

However, the returned address must be “accessible” for it to make any sense

- Remember, local variables in the called function will vanish, after the call returns
- So, returning the address to such a variable can be disastrous !!

Some additional points

We can use the same concept while returning values as well…

- … i.e., instead of returning a value, we can also return an address

However, the returned address must be “accessible” for it to make any sense

- Remember, local variables in the called function will vanish, after the call returns
- So, returning the address to such a variable can be disastrous !!
- A good candidate for returning the address, is of any memory block allocated via `malloc()`

Some additional points

We can use the same concept while returning values as well…

- … i.e., instead of returning a value, we can also return an address

However, the returned address must be “accessible” for it to make any sense

- Remember, local variables in the called function will vanish, after the call returns
- So, returning the address to such a variable can be disastrous !!
- A good candidate for returning the address, is of any memory block allocated via `malloc()`

Arrays are always passed by reference in C

- This follows from the fact that array names are, in fact, pointer constants
- Remember: the expression `arr` (where `arr` is an array’s name) is equivalent to `&arr[0]`

Example – The `trim()` function… again

```c
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

void trim(char str[])
{
        int len = strlen(str), beg = 0, end = len-1, ctr = 0;
        while(beg <= end && str[beg] == ' ')
                beg++;
        while(end >= beg && str[end] == ' ')
                end--;
        for(ctr = 0; beg <= end; ctr++, beg++)
                str[ctr] = str[beg];
        str[ctr] = '\0';
}

int main()
{
        char str[] = "  Hello  ";
        printf("The initial array is \"%s\"\n", str);
        trim(str);
        printf("The array after trimming is \"%s\"\n", str);
}
```

Example – The `trim()` function… again

```c
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

void trim(char str[])
{
        int len = strlen(str), beg = 0, end = len-1, ctr = 0;
        while(beg <= end && str[beg] == ' ')
                beg++;
        while(end >= beg && str[end] == ' ')
                end--;
        for(ctr = 0; beg <= end; ctr++, beg++)
                str[ctr] = str[beg];
        str[ctr] = '\0';
}

int main()
{
        char str[] = "  Hello  ";
        printf("The initial array is \"%s\"\n", str);
        trim(str);
        printf("The array after trimming is \"%s\"\n", str);
}
```

Even though we don't pass any addresses here explicitly…

Example – The `trim()` function… again

```c
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

void trim(char str[])
{
        int len = strlen(str), beg = 0, end = len-1, ctr = 0;
        while(beg <= end && str[beg] == ' ')
                beg++;
        while(end >= beg && str[end] == ' ')
                end--;
        for(ctr = 0; beg <= end; ctr++, beg++)
                str[ctr] = str[beg];
        str[ctr] = '\0';
}

int main()
{
        char str[] = "  Hello  ";
        printf("The initial array is \"%s\"\n", str);
        trim(str);
        printf("The array after trimming is \"%s\"\n", str);
}
```

Even though we don't pass any addresses here explicitly…

… the changes made in the called function can be observed in the calling function

Example – The `trim()` function… again

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 8$ gcc TrimString.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 8$ ./a.out
The initial array is "  Hello  "
The array after trimming is "Hello"
saurabh@saurabh-VirtualBox:~/C/examples/Week 8$
```

Example – The `trim()` function… again

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 8$ gcc TrimString.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 8$ ./a.out
The initial array is "  Hello  "
The array after trimming is "Hello"
saurabh@saurabh-VirtualBox:~/C/examples/Week 8$
```

Now, the changes made in the called function…

Some additional points

We can use the same concept while returning values as well…

- … i.e., instead of returning a value, we can also return an address

However, the returned address must be “accessible” for it to make any sense

- Remember, local variables in the called function will vanish, after the call returns
- So, returning the address to such a variable can be disastrous !!
- A good candidate for returning the address, is of any memory block allocated via `malloc()`

Arrays are always passed by reference in C

- This follows from the fact that array names are, in fact, pointer constants
- Remember: the expression `arr` (where `arr` is an array’s name) is equivalent to `&arr[0]`

If you are coming from C++ background, you may find the terminology a little confusing

- In C++, whatever we studied, is known as *pass by pointers*, whereas *pass by reference* is something else
- But in C, there is no concept of a reference, thus, pass by pointers is *a way to emulate* pass by reference

Homework !!

The terminology problem does not really have a perfect answer…

- It is true that C only allows passing parameters by value, and thus, using pointers, is kind of a hack
- But that doesn't mean that it cannot by called "pass by reference"
- Read the discussions on this interesting thread: https://stackoverflow.com/questions/59048556/no-call-by-reference-in-c