CS1100 - Lecture 23

Instructor : Jasine Babu

Teaching Assistants : Nikhila K N, Veena Prabhakaran

Call by value using pointers

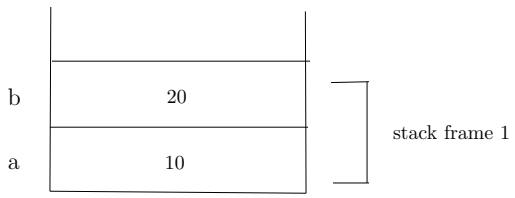
Recall that, C language supports only call by value for passing parameters. In this method, when we invoke a function, the values of the actual parameters are copied respectively to the locations of the formal parameters. Consider the following example.

An incorrect way of doing swap

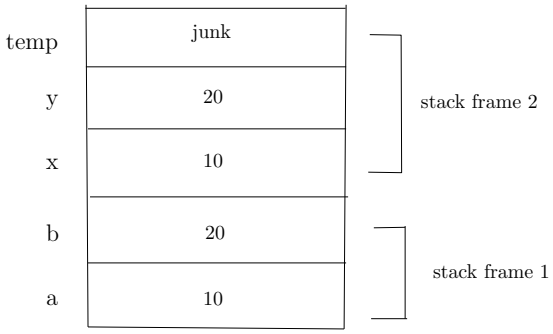
```
#include <stdio.h>
void swap_try(int, int);
int main()
{
  int a=10, b=20;
  printf("a=%d, b=%d\n", a, b);
  swap_try(a, b);
  printf("a=%d, b=%d\n", a, b);
  return(0);
}

void swap_try(int x, int y)
{
  int temp;
  temp=x;
  x=y;
  y=temp;
}
```

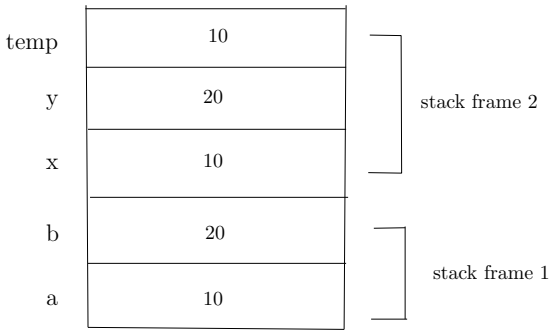
Initially, the program control is in the `main()` function and the current stack frame in memory is as follows.



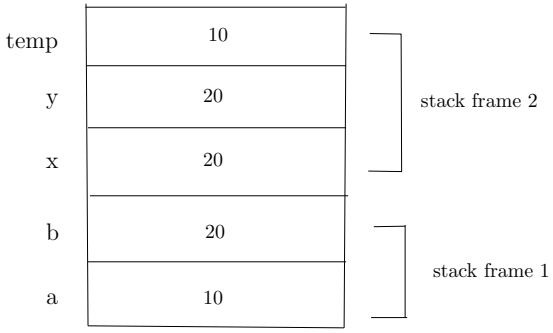
After executing the first `printf()` in the `main()` function, the `swap_try(a,b)` function is invoked and a new stack frame for the function `swap_try()` is created; the values of formal parameters `a` and `b` get copied to the locations of the variables `x` and `y` in the new stack frame. After this, program control transfers to the beginning of the function `swap_try()`.



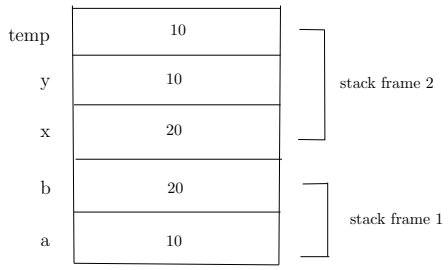
After executing the instruction `temp=x;`, the contents of the stack frame are as follows.



The stack frame in memory after executing the instruction `x=y;` is shown below.

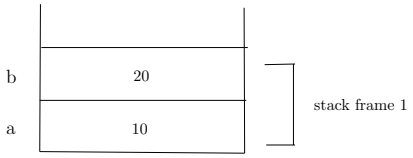


After executing the instruction `y=temp;`, the stack frame in memory is as follows.



In the next step, the function finishes its execution. So the frame 2 will be destroyed and the control goes back to the `main()` function.

The memory state diagram at this point is as follows.



When the next line is executed, the values of `a` and `b` will be displayed for a second time. The output of the program is shown below.

```
a=10, b=20
a=10, b=20
```

Now, we can see that nothing has happened to the variables `a` and `b`. The change was made only to `x` and `y` which had their values copied from `a` and `b` respectively. This change does not affect the values of `a` and `b` in the stack frame of the `main()` function. The changed variables also get destroyed when the function call returns.

Note that, even if we use the names `a` and `b` for formal parameters in the definition of the function `swap_try()` as given below, the program works exactly the same way as the previous program.

A second incorrect way of swap

```
#include <stdio.h>
void swap_try(int, int);
int main()
{
  int a=10, b=20;
  printf("a=%d, b=%d\n", a, b);
  swap_try(a, b);
  printf("a=%d, b=%d\n", a, b);
  return(0);
}

void swap_try(int a, int b)
{
  int temp;
  temp=a;
  a=b;
  b=temp;
}
```

Even though the variable names in `main()` and `swap_try()` are the same, when the function call `swap_try(a,b)` is made from `main()`, a new stack frame is created for `swap_try()` with new locations for storing the values of the formal variables `a` and `b` and the local variable `temp`. The modifications made inside `swap_try()` affect only the newly created variables `a` and `b` in the stack frame of `swap_try()`. When the function `swap_try()` returns to `main()`, the stack frame of `swap_try()` gets deleted and the modifications done to the new variables `a` and `b` are lost as in the first example.

Correcting the swap function

Since we have many situations where we would need to make modifications to data by passing parameters to functions, we will go for an indirect way of modifying data values by passing their addresses as formal parameters to functions. (However, it should emphasized that what we use is call by value itself.)

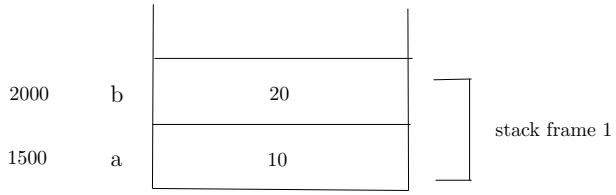
If we want to get the values of variables `a` and `b` declared in `main()` interchanged using a `swap()` function, we can use the following method.

Correct way of doing swap

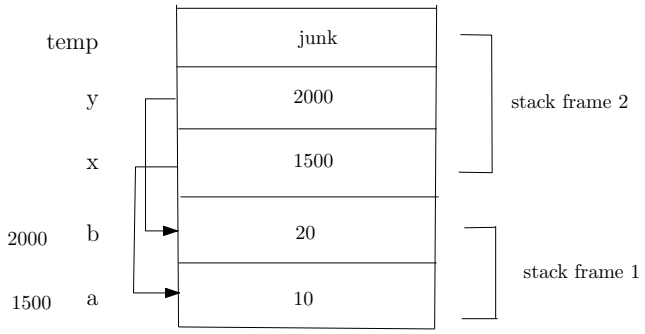
```
#include <stdio.h>
void swap(int*, int*);
int main()
{
  int a=10, b=20;
  printf("a=%d, b=%d\n", a, b);
  swap(&a, &b);
  printf("a=%d, b=%d\n", a, b);
  return(0);
}
void swap(int *x, int *y)
{
  int temp;
  temp=*x;
  *x=*y;
  *y=temp;
}
```

In the above program, the formal parameters of the function `swap()` are two pointers to integers (`int *x` and `int *y`). From the `main()` function, when the `swap()` function is invoked, the parameters passed are the addresses of the integer variables `a` and `b`. When the `swap(&a,&b);` function call is made, the formal parameters `x` and `y` (which are pointer variables) get their values from the values of the corresponding actual parameters `&a` and `&b`. Note that, the types of the formal parameters and the corresponding actual parameters match.

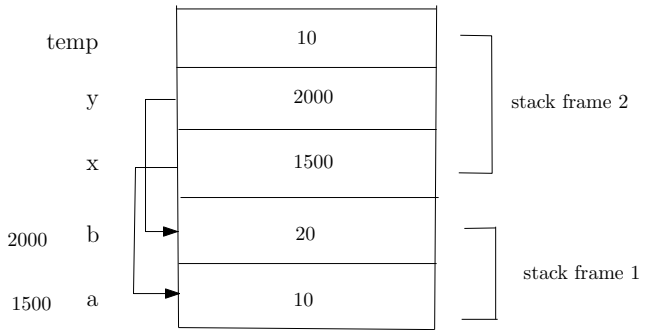
Consider the execution of the above program. The memory state diagram just before the function call is shown below.



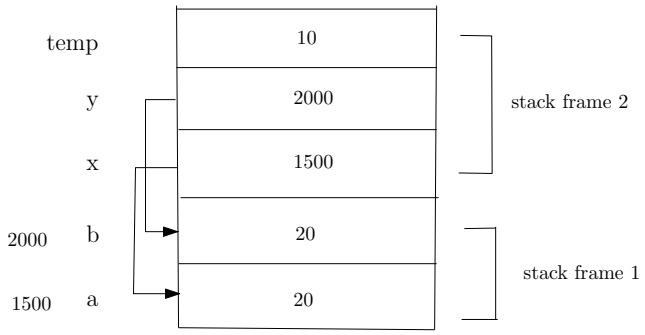
When the function call `swap(&a,&b)` is executed, a new stack frame for the function `swap()` is created in memory. This new stack frame will have locations for storing the pointer variable `x` and `y` and the integer variable `temp`. The integer pointer variable `x` gets the value of `&a` and the integer pointer variable `y` gets the value of `&b`. Initially, the value of variable `temp` is junk. The memory state diagram after the function call is as follows.



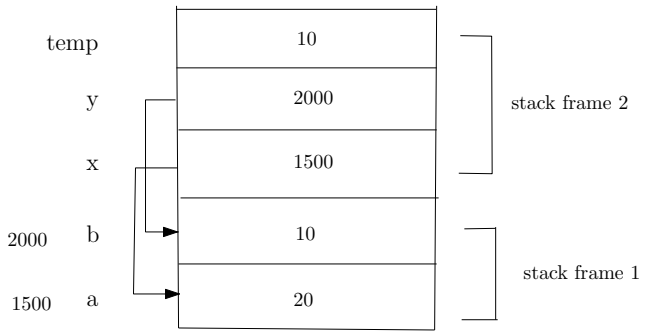
Since `x` stores the address `1500`, the value of the expression `*x` is the content of the location `1500` which is `10`. After executing the instruction `*temp=*x;`, the variable `temp` gets the value `10`. The memory state diagram at this stage is as follows.



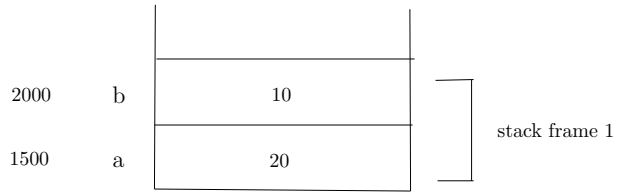
After that, the instruction `*x=*y;` is executed. Since `y` stores the address `2000`, the value of the expression `*y` is the content of the location `2000` which is `20`. The expression `*x` on the left hand side of the instruction refers to the location whose address is `1500` (which is the value of `x`). Therefore, after executing the instruction `*x=*y;`, the value `20` gets stored in the location with address `1500`. The contents of memory locations at this point of execution is shown below.



When the instruction `*y=temp;` get executed, the variable `temp` is copied to the location whose address is `2000` (which is the value of `y`). The contents of the memory locations at this point is as follows.



After executing the instruction `*y=temp;`, the function `swap()` finishes its execution. Now, the frame 2 is destroyed and the program control returns to the `main` function.



In the next line, the updated values of `a` and `b` will be displayed. The output of the above program is given below.

```
a=10, b=20
a=20, b=10
```

Thus, if we want to change the values of some data using parameter passing, we should pass the address of the data to be changed as parameter to the function. If `x` is a formal parameter in the definition of a function `f()`, any modifications done to the value of `x` get lost after the function call, because when the function `f()` returns to the calling function, the frame containing `x` gets destroyed. However, if `x` is a pointer, any modifications done to `*x` will persist even after returning from the function `f()`.

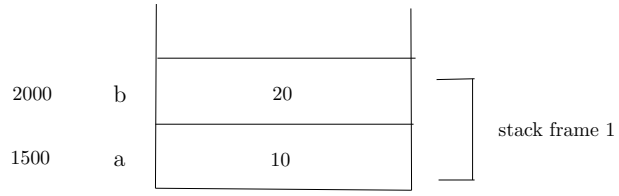
Note that, even when the formal parameter `x` of a function `f()` is a pointer, any modifications done to the value of `x` will be lost once the function call `f()` returns to the calling function. The following program illustrates this.

A third incorrect definition of swap

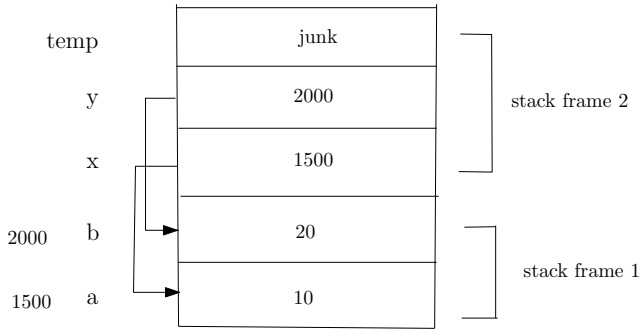
```
#include <stdio.h>
void swap_try(int*, int*);
int main()
{
  int a=10, b=20;
  printf("a=%d, b=%d\n", a, b);
  swap_try(&a, &b);
  printf("a=%d, b=%d\n", a, b);
  return(0);
}

void swap_try(int *x, int *y)
{
  int *temp;
  temp=x;
  x=y;
  y=temp;
}
```

The memory state diagram just before the function call is as follows.

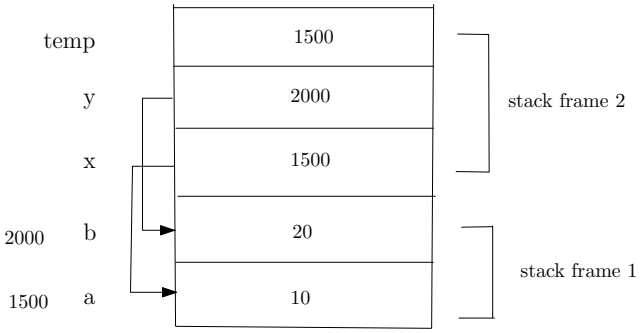


After the function call, the new frame for the function `swap_try()` is created and the current memory state diagram is as follows

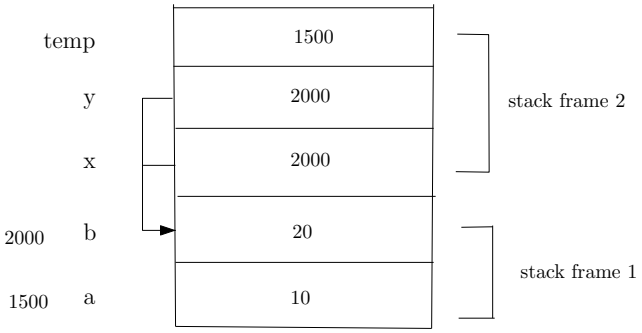


In the above program, the variables `temp, x` and `y` are integer type pointer variables. They can hold the addresses of an integer variables. After executing the instruction

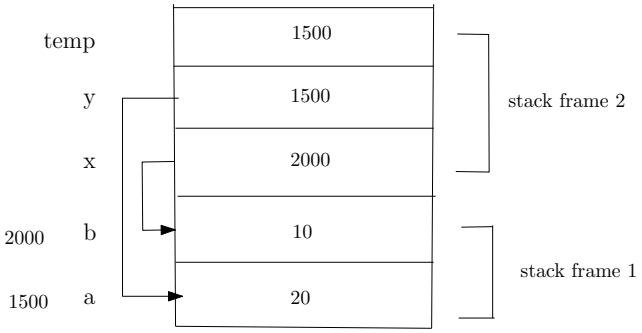
`temp=x;`, the pointer variable `temp` gets the value of `x`, which is `1500`. The memory state diagram after executing the instruction `temp=x;` is shown below.



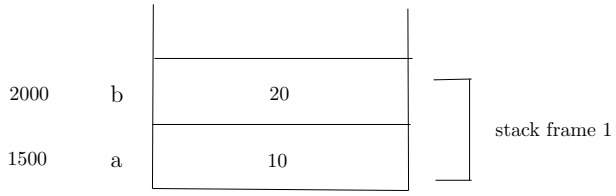
After that, the instruction `x=y;` is executed and `x` gets the value of `y` which is `2000`. The memory state diagram at this point is shown below.



Finally, the instruction `y=temp` gets executed and the pointer variable `y` gets the value of `temp` which is `1500`. The contents of the memory locations at this point is as follows.



After executing the instruction `y=temp;`, the function `swap_try()` finishes its execution. Note that, during the execution of `swap_try()`, the variables `a` and `b` in the stack frame of `main()` are not at all modified.



When the function `swap_try()` returns to the `main()` function, frame 2 will be destroyed. The value of `a` and `b` will be displayed in the next line. The output of the above program is given below.

```
a=10, b=20
a=10, b=20
```