Functions, Memory Layout - stack, and Arrays

Jamal Hussein

jah1g12@ecs.soton.ac.uk

Electronics and Computer Science

November 20, 2015

Outline

1. Functions
2. Calling a Function
3. Recursion
4. Memory Layout - Stack
5. Arrays
   - 2D Arrays

Defining Functions

- General form of a function definition is:

```
return_type function_name( parameter list )
{
          body of the function
}
```

- The `return_type` is the type the function returns
- Some functions do not return values, in this case `void` is used
- The function name and the parameter list together constitute the function signature
- a function may contain no parameters
- The function body contains a collection of statements that define what the function does

Example

```
/* function returning the max between two numbers */
int max(int num1, int num2)
{
        /* local variable declaration */
        int result;
        if (num1 > num2)
                result = num1;
        else
                result = num2;

        return result;
}
```

Function Declarations

- A function declaration tells the compiler about a function name and how to call the function
- The actual body of the function can be defined separately

```
return_type function_name(parameter list);
```

- for example:

```
int max(int num1, int num2);
```

- the parameters can be dropped in function declaration:

```
int max(int, int);
```

Calling a Function

```
#include <stdio.h>
/* function declaration */
int max(int num1, int num2);
int main (){
        int a = 100, b = 200, ret;
        ret = max(a, b); /* a and b are arguments*/
        printf( "Max value is : %d\n", ret);
        return 0;
}
/* function definition */
int max(int num1, int num2){
        int result;
        if (num1 > num2) result = num1;
        else result = num2;
        return result;
}
```

call by value

- The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function
- changes made to the parameters inside the function have no effect on the arguments

call by value

```
void swap(int x, int y);
int main (){
        int a = 100, b = 200;
        printf("Before swap, value of a : %d\n", a );
        printf("Before swap, value of b : %d\n", b );
        swap(a, b);
        printf("After swap, value of a : %d\n", a );
        printf("After swap, value of b : %d\n", b );
        return 0;
}
void swap(int x, int y){
        int temp = x; /* save the value of x */
        x = y; /* put y into x */
        y = temp; /* put x into y */
}
```

call by reference

- The **call by reference** method of passing arguments to a function copies the address of an argument into the formal parameter
- Inside the function, the address is used to access the actual argument used in the call
  - changes made to the parameter affect the passed argument

  To pass by reference, argument pointers are passed to the functions just
- the function parameters need to be declared as **pointer** types

call by reference

```
void swap(int *x, int *y);
int main (){
        int a = 100, b = 200;
        printf("Before swap, value of a : %d\n", a);
        printf("Before swap, value of b : %d\n", b);
        swap(&a, &b);
        printf("After swap, value of a : %d\n", a);
        printf("After swap, value of b : %d\n", b);
        return 0;
}
void swap(int *x, int *y){
        int temp = *x; /* save the value of x */
        *x = *y; /* put y into x */
        *y = temp; /* put x into y */
}
```

Recursion

- C supports Recursion: a function to call itself
- you must define an exit condition
  - otherwise the function will go in infinite loop
- recursive functions are used to solve problems such as
  - finding factorial of a number (n!)
  - generating Fibonacci series

factorial

```c
#include <stdio.h>
int factorial(unsigned int i)
{
        if(i <= 1) /* exit condition */
        return 1;
        return i * factorial(i - 1);
}
int main()
{
        int n = 15;
        printf("%d! is %d\n", n, factorial(n));
        return 0;
}
```

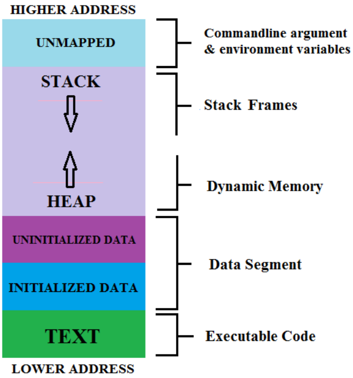
factorial

```c
#include <stdio.h>

int factorial(unsigned int i)
{
        return (i <= 1) ? 1 : i * factorial(i-1);
}

int main()
{
        int i = 15;
        printf("%d! is %d\n", i, factorial(i));
        return 0;
}
```

Fibonacci Series

```
#include <stdio.h>
int fibonacci(int i) {
        if(i == 0) return 0;
        if(i == 1) return 1;
        return fibonacci(i-1) + fibonacci(i-2);
}
int main() {
        int i;
        for (i = 0; i < 10; i++) {
                printf("%d\t%n", fibonacci(i));
        }
        return 0;
}
```

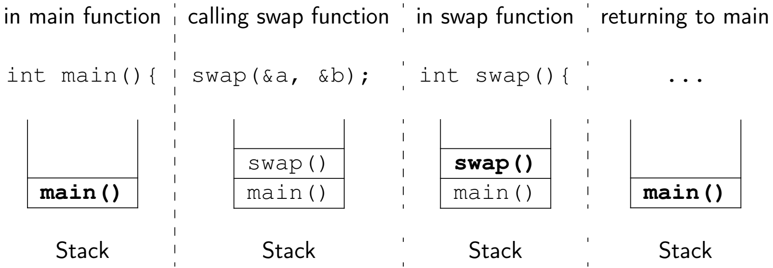
Memory Layout - Stack



Memory Layout - Stack

- The stack segment is the area where local variables are stored
    - local variable are declared in every function including main() in C program
- When we call any function a **stack frame** is created
- function returns, stack frame is destroyed including all local variables of that particular function
- Stack frame contain some data like **return address**, arguments passed to the function, local variables,. . .
- A “stack pointer (SP)” keeps track of stack by each push and pop operation onto it

Memory Layout - Stack



in main function

`int main(){`

main()

Stack

calling swap function

`swap(&a, &b);`

swap()

main()

Stack

in swap function

`int swap(){`

swap()

main()

Stack

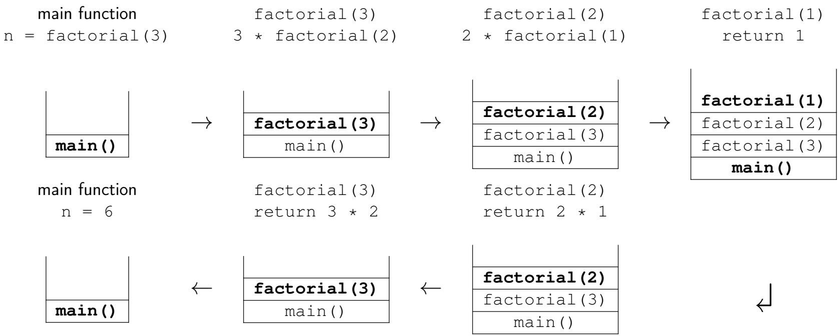
returning to main

...

main()

Stack

Memory Layout - Stack



main function
n = factorial(3)

factorial(3)
3 * factorial(2)

factorial(2)
2 * factorial(1)

factorial(1)
return 1

main()

→

factorial(3)
main()

→

factorial(2)
factorial(3)
main()

→

factorial(1)
factorial(2)
factorial(3)
main()

main function
n = 6

factorial(3)
return 3 * 2

factorial(2)
return 2 * 1

main()

←

factorial(3)
main()

←

factorial(2)
factorial(3)
main()

↲

Arrays

- Array a fixed-size sequential collection of elements of the same type
- An array is used to store a collection of data of the same type
- All arrays consist of contiguous memory locations
  - The lowest address corresponds to the first element and the highest address to the last element
- Declaring arrays: `double balance[10];`

Arrays

- Initializing array:

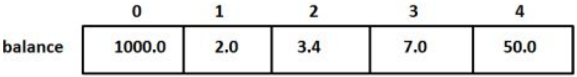
```
double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

- we can ommit the size:

```
double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

- Accessing an array:

```
balance[4] = 50.0;
double salary = balance[3];
```

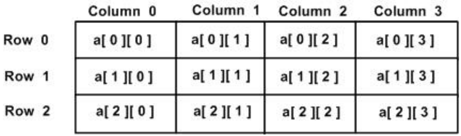


	0	1	2	3	4
balance	1000.0	2.0	3.4	7.0	50.0

Two-dimensional Arrays

- The simplest form of multidimensional array is the two-dimensional array
- A two-dimensional array is an array of one-dimensional arrays
- Declaring 2D arrays:

```
int a[3][4];
```



	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Two-dimensional Arrays

- Initializing 2D arrays:

```
int a[3][4] = {
              {0, 1, 2, 3} ,
              {4, 5, 6, 7} ,
              {8, 9, 10, 11}
};
```

- or we can omit the brackets

  `int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};`

- accessing 2D arrays:

  ```
  int val = a[2][3];
  a[1][1] = -5;
  ```

Two-dimensional Arrays

```
#include <stdio.h>

int main () {

        /* an array with 5 rows and 2 columns*/
        int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
        int i, j;

        /* output each array element's value */
        for ( i = 0; i < 5; i++ ) {
                for ( j = 0; j < 2; j++ ) {
                        printf("a[%d][%d] = %d\n",
                                i,j, a[i][j] );
                }
        }

        return 0;
}
```

Two-dimensional Arrays

- three ways to pass an array to functions
  1. as a pointer:
     `void myFunction(int *param) {...}`
  2. as a sized array:
     `void myFunction(int param[10]) {...}`
  3. as a unsized array:
     `void myFunction(int param[]) {...}`

Passing arrays to functions

```
#include <stdio.h>
double getAverage(int arr[], int size);
int main () {
        int balance[5] = {1000, 2, 3, 17, 50};
        double avg;
        avg = getAverage( balance, 5 ) ;
        printf( "Average value is: %f\n", avg );
        return 0;
}
double getAverage(int arr[], int size) {
        int i;
        double avg, sum;
        for (i = 0; i < size; ++i) {
                sum += arr[i];
        }
        avg = sum / size;
        return avg;
}
```