Functions: function invocation & call-by-value, recursion

Function invocation and call-by-value

A C program consists of one or more function definitions, and `main()` function must be one of them. Program execution always begins with `main()`. When a program executes a function name, the function is called, or invoked. Then, program execution transfers to that function. After the function execution finishes, the program resumes execution in the calling environment. Functions are called by writing their name together with the appropriate list of arguments within parentheses. Usually, these arguments match in number and type with the parameters in the function's parameter list. **All arguments are passed by-value which is called "call-by-value".** Since the previous sentence is very important, I have emphasized it in bold. This means that each argument is evaluated, and its value is passed. *Thus, if a variable is passed to a function, we are only passing the value of that variable.* But the stored value of that variable in the calling environment will not be changed. To illustrate the call-by-value concept, consider the following program that calculates the average of three integers using a function.

```
#include<stdio.h>
double calav(int,int,int);  // function prototype
int main()
{
int a=5,b=3,c=7;
double av;
av=calav(a,b,c);          // function involcation
printf("Average=%.2f\n",av);
return 0;
}
double calav(int a,int b,int c)
{
 int s;
 s=a+b+c;
 return s/3.0;
}
```

Listing 39: C program "avgi.c"

The output of the code is `Average=5.00`. The declaration `int a=5,b=3,c=7;` causes the values `5`, `3` and `7` to be stored in the `int` variables `a`, `b` and `c` respectively. Each of these `int` variables is stored in a unique address in memory. We can think of memory as a very long, thin tape in which the identifiers are stored in unique addresses. For example, to see the address of the variable `a`, we include the `printf("%p\n",&a);` statement. Its output on my macOS is `0x16ce03208`, in hexadecimal format. Of course, the address does not remain the same and it

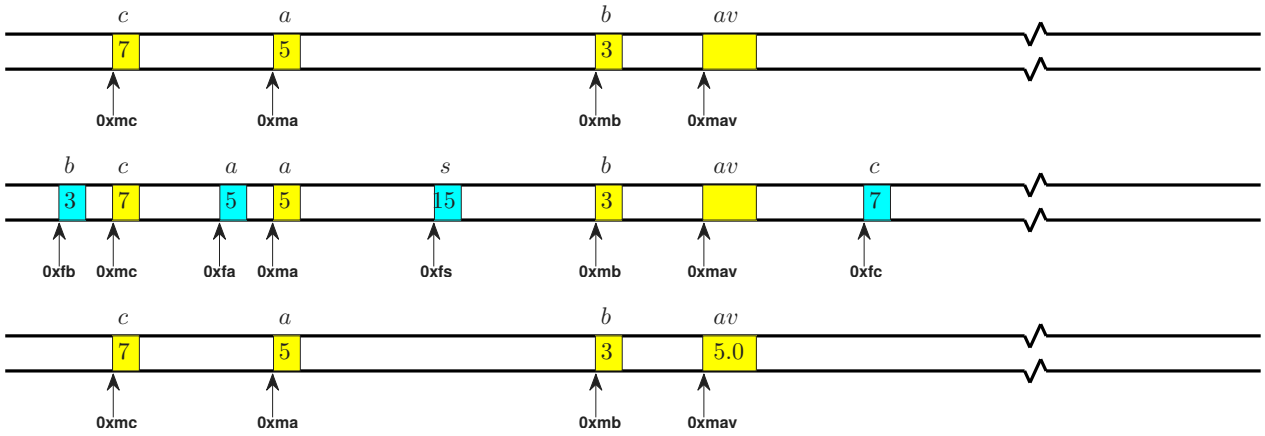

Figure 1: Memory arrangement for Listing 39

may change if we run the code later. After the declaration in `main()`, the `int` variables are stored in (**hypthetical**) addresses as shown in the top panel of Figure 1. There is no particular order in which these are stored in the memory. I have used yellow-filled colour for identifiers of `main()`. Also, (hypothetical) addresses start with `0xm` for identifiers in `main()`. For example, identifier `a` is stored in address `0xma`. These `int` variables typically take 4 bytes of memory. Similarly, the `double` variable `av` is stored in address `0xmav` and a `double` variable typically takes 8 bytes of memory. Note that the content of `av` is undefined when it is declared. The function call `av=calav(a,b,c);` is equivalent to `av=calav(5,3,7);` and thus we are passing values `5, 3` and `7`. The `int` variables `a, b` and `c`, defined within parentheses in the function `calav()` declaration, are local to the function `calav()` and they received the value `5, 3` and `7` respectively. They are also stored in memory, shown as cyan-filled colour in the middle panel of Figure 1. Also, the `int` variable `s` declared within `calav()` is also local to `calav()`. The arithmetic statement `s=a+b+c;` causes `15` to be assigned to `s`. The function `calav()` returns `5.0` to the calling function `main()` and `5.0` is assigned to `av`. Now the work of the function `calav()` is finished, and the variables related to `calav()` (cyan-filled) are no longer valid. The last panel of Figure 1 shows the memory configuration after the call to `calav()` finishes.

Let us consider another program in which we intend to swap two integer values using a function. However, as we see shortly, the function fails to do the job. The memory configuration after the declaration `int a=5,b=3;` in `main()` is shown in the top panel in Figure 2. The identifiers of the `main()` are stored in locations marked in yellow. The function call is made via `wrngswap(a,b);` is equivalent to `wrngswap(5,3);`. Note that identifiers `a, b`, that occur inside parentheses of function definition, and `t` are local to `wrngswap()`. These are stored in locations marked cyan. Also, the `int` variables `a` and `b` inside the parentheses of the function definition acquire values `5` and `3` respectively. When the declaration `int t;` is made inside the `wrngswap()`, the corresponding configuration is shown in the second panel in Figure 2. The swapping inside `wrngswap()` produces a swap between local `a` and `b` as shown in the third panel in Figure 2. Note that writing `a=b;` followed by `b=a;` does not swap values between `a,b`.

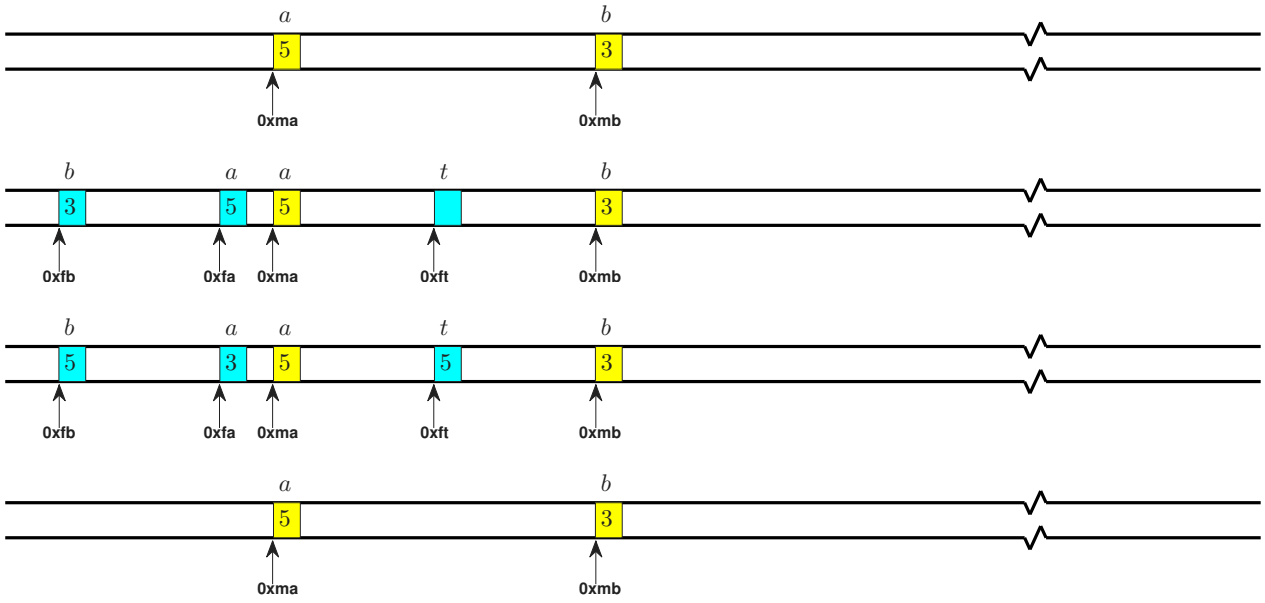

Figure 2: Memory arrangement for Listing 40

Since `a=b` results in both `a` and `b` having value `3`. Hence, we need an auxiliary variable (`t` here) that holds the value of one of the variables. Note that the `int` variables `a` and `b` in `main()` remain unaffected. After execution in `wrngswap()` finishes, the control returns to the calling function `main()`, and the variables related to `wrngswap()` (cyan-filled) are no longer valid. The last panel in Figure 2 shows the memory configuration after the call to `wrngswap()` finishes. Clearly, the function's purpose is not accomplished.

```
#include<stdio.h>
void wrngswap(int,int);      // function prototype
int main()
{
int a=5,b=3;
printf("Before swap: a=%d  b=%d\n",a,b);
wrngswap(a,b);
printf("After swap : a=%d  b=%d\n",a,b);
return 0;
}

void wrngswap(int a,int b)
{
 int t;
 t=a;
 a=b;
 b=t;
```

```
}
```

Listing 40: C program "cwrnswp.c"

The output of the code is

```
Before swap: a=5  b=3
After swap : a=5  b=3
```

From the above discussion, the reason for the failure of the swap is clear. Since, we have sent the values of `a` and `b` only, the function `wrngswap()` can't access the original `a` and `b` of the `main()`. To make an analogy, you can think of original and photocopy. The function receives the photocopies of the original `a` and original `b`. The function can change the photocopies as it likes but the originals in the `main()` are kept safe. To change the originals in the `main()`, the function need to know where it is stored, i.e., the address `0xma` of `a` and the address `0xmb` of `b`. This is the subject of pointer that we will discuss later.

Recursion

A recursive program in C is one where a function calls itself to solve a smaller version of a problem until a specific base case (stopping condition) is met. For example, consider the factorial function, which is defined as

$$n! = \begin{cases} n \times (n-1)! & n \geq 2 \\ 1 & n = 0, 1. \end{cases}$$

Thus, to find 4!, we note that

$$\begin{aligned} 4! &= 4 \times 3! \\ 3! &= 3 \times 2! \\ 2! &= 2 \times 1! \\ 1! &= 1 \end{aligned}$$

Clearly, $1! = 1$ works as base case. Now we go backwards:

$$\begin{aligned} 2! &= 2 \times 1 = 2 \\ 3! &= 3 \times 2 = 6 \\ 4! &= 4 \times 6 = 24 \end{aligned}$$

We write a function `rfactorial()` that takes an integer argument and returns an integer. The complete program is given below.

```
#include<stdio.h>
int rfactorial(int);     //function prototype
int main()
```

```
{
 int n;
 printf("Enter nonnegative integer n:");
 scanf("%d",&n);
 if(n<0)             //validity of input
 {
  printf("Input must be non-negative\n");
  return 0;
 }
 printf("Factorial(%d)=%d\n",n,rfactorial(n)); //function invocation
 return 0;

}
int rfactorial(int n)
{
 if(n==0 || n==1)
  return 1;
 return n*rfactorial(n-1);
}
```

Listing 41: C program "crecfct.c"

Below is some input-output of the program.

```
Enter nonnegative integer n:4
Factorial(4)=24
Enter nonnegative integer n:-2
Input must be non-negative
Enter nonnegative integer n:0
Factorial(0)=1
```

If we trace the function call `rfactorial(4)`, we see that

i. `rfactorial(4)` returns 4× `rfactorial(3)`

ii. `rfactorial(3)` returns 3× `rfactorial(2)`

iii. `rfactorial(2)` returns 2× `rfactorial(1)`

iv. `rfactorial(1)` returns 1 (base case)

v. `rfactorial(2)` becomes 2

vi. `rfactorial(3)` becomes 6

vii. `rfactorial(4)` becomes 24

Note that the input must be checked for validity. This is because the function `rfactorial()` will be called infinitely many times when the input is negative. Of course, the validity could have been checked within the function as well. Note that calling a recursive function is CPU-intensive, since the system must keep track of the execution trace. To avoid this, we write a function using iteration, which uses loops such as `for` and `while`. Below is a non-recursive function `nrfactorial()` which could be used in place of `rfactorial()`.

```
int nrfactorial(int n)
{
 int i,s=1;
 for(i=1;i<=n;i++)
 {
  s *=i;
 }
 return s;
}
```

Next, we implement another program that calculates the Fibonacci number f_n, which is defined by

$$f_0 = 0,\ f_1 = 1,\ f_n = f_{n-1} + f_{n-2}, \quad n \geq 2.$$

First, we write this program using a recursive function `rfibo()`. We also calculate the number of times this recursive function calls itself. To implement the last task, we use a **global variable**. A global variable in C is a variable declared outside of all functions, usually at the top of the program before the `main()` function. It has a global scope, meaning it can be accessed and modified by any function within the program.

```
#include<stdio.h>
int rfibo(int);     //function prototype
int count=0;       // global variable initialization
int main()
{
 int n;
 printf("Enter nonnegative integer n:");
 scanf("%d",&n);
 if(n<0)             //validity of input
 {
  printf("Input must be non-negative\n");
  return 0;
```

```
 }
 printf("Fibo(%d)=%d\n",n,rfibo(n)); //function invocation
 printf("No of calls=%d\n",count);
 return 0;
}
int rfibo(int n)
{
 count++;
// printf("Calling rfibo counter %d with n=%d\n",count,n);
 if(n<=1)
 {
  return n;
 }
 return rfibo(n-1)+rfibo(n-2);
}
```

Listing 42: C program "crfibo.c"

Below is some sample input-output.

```
Enter nonnegative integer n:23
Fibo(23)=28657
No of calls=92735
Enter nonnegative integer n:5
Fibo(5)=5
No of calls=15
```

As can be seen from the output, the number of times the function calls itself becomes very large. The trace of calls for `n=5` is shown in Figure 3. With `n=5`, the first call is made `rfibo(5)` which returns `rfibo(4)+rfibo(3)`. However, once it encounters `rfibo(4)`, the function calls itself with `n=4` (see the top branch in Figure 3). Once the call to `rfibo(4)` finishes, only then `rfibo(3)` will be called and the result returned by `rfibo(3)` will be added to that of `rfibo(4)` to arrive at the finite result return by `rfibo(5)`. Clearly, the call to `rfibo(4)` again proceeds as in the case of `rfibo(5)`. The function returns a value `0` for `n=0` and a value `1` for `n=1`. The values in blue font denotes the values returned by the functions. The values in red font denote the sequences in function call. To see the sequence of calls, you may uncomment the statement commented in function definition of `rfibo()`. Then the following output conforms the Figure 3.

```
Enter nonnegative integer n:5
Calling rfibo counter 1 with n=5
```

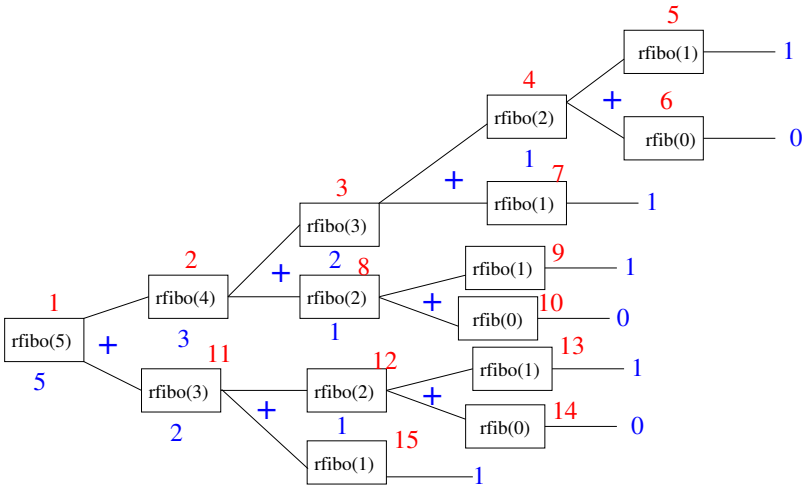

Figure 3: Recursive call to the function rfibo() with n=5. Values in blue are returned by function and red values denote sequence of calls.

```
Calling rfibo counter 2 with n=4
Calling rfibo counter 3 with n=3
Calling rfibo counter 4 with n=2
Calling rfibo counter 5 with n=1
Calling rfibo counter 6 with n=0
Calling rfibo counter 7 with n=1
Calling rfibo counter 8 with n=2
Calling rfibo counter 9 with n=1
Calling rfibo counter 10 with n=0
Calling rfibo counter 11 with n=3
Calling rfibo counter 12 with n=2
Calling rfibo counter 13 with n=1
Calling rfibo counter 14 with n=0
Calling rfibo counter 15 with n=1
Fibo(5)=5
No of calls=15
```

As seen in the above discussion, the code with recursion for Fibonacci number involves a lot of overhead due to keeping track of function calls and outputs. The following non-recursive function `nrfibo()` can be used in place of `rfibo()`.

```
int nrfibo(int n)
{
 int i,f0=0,f1=1,fn;

 if(n<=1)
```

```
 {
  return n;
 }

 for(i=2;i<=n;i++)
 {
  fn=f0+f1;
  f0=f1;
  f1=fn;
 }
 return fn;
}
```

Recursion can offer concise code but uses more memory and can be slower than iteration. It is often used in tree/graph traversals, sorting, and related operations.

Return to Lecture Topics