DVA104

Data Structures, Algorithms, and Program Design.

Gabriele Capannini

gabriele.capannini@mdh.se

Mälardalen University

July 17, 2018

Outline

- Repetition
    - Memory
    - Functions
    - Pointers
- Recursion (functions that call themselves)
- Double pointers (pointers to pointers)

Repetition

- Function calls
  - **Actual Parameters** are the variables passed from the calling function.
  - **Formal Parameters** are the variables written in the function declaration:
    - they belong to the function and are created upon entry into the function and destroyed upon exit;
    - they are initialized with the value passed by the actual parameters. Hence, any update made to the formal parameter does not affect the actual one.
- Pointers as parameters
  - Allows you to change the value pointed (not the pointer by itself) by the argument passed from the calling function.

Repetition (continued)

Parameters (or arguments) in practice:

```
void foo(int a) {
  printf("%d", a+1);
}
int main(void) {
  int x = 10;
  foo(x);
  return 0;
}
```

In the above example, `x` is the actual parameter, while `a` is the formal parameter.

Note that the formal and actual parameters can have the same name (e.g., the formal parameter is called `x` as well). Even in this case, they are different variables (i.e., they are stored in different locations in memory).

Repetition (continued)

In general, **stacks** are arrays managed with a LIFO (Last In, First Out) policy which means that a new item is added at the end of the array as well as the extraction of an item.

In a program many functions coexist at the same time, but (for the moment) we assume that only one at a time is active and running.

The **Call Stack** is used to keep track of the active function, its variables, and where it returns the control when it ends.

- The active function is the one on the top of the stack.
- A function call corresponds to allocate (on the top of the stack) the memory for storing (at least) the variables declared inside the function.
- When a function ends (e.g., return a value), the memory allocated by the function call is removed from the stack.

Repetition (continued)

```
int h(void) {
  return 0;
}
int g(void) {
  return 1+h();
}
int f(void) {
  return 1+g();
}
int main(void) {
  int x = 1+f();
  return 0;
}
```

Call Stack:

main { `x=?`

The first function to be called is always `main`

Repetition (continued)

```
int h(void) {
  return 0;
}
int g(void) {
  return 1+h();
}
int f(void) {
  return 1+g();
}
int main(void) {
  int x = 1+f();
  return 0;
}
```

Call Stack:

f { $\varnothing$

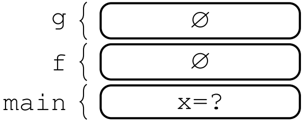
main { x=?

. . . Then `main` calls `f()`

Repetition (continued)

```
int h(void) {
  return 0;
}
int g(void) {
  return 1+h();
}
int f(void) {
  return 1+g();
}
int main(void) {
  int x = 1+f();
  return 0;
}
```

Call Stack:

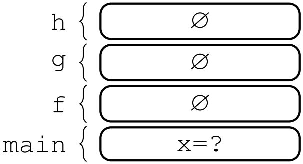


. . . Then `f` calls `g()`

Repetition (continued)

```c
int h(void) {
  return 0;
}
int g(void) {
  return 1+h();
}
int f(void) {
  return 1+g();
}
int main(void) {
  int x = 1+f();
  return 0;
}
```

Call Stack:

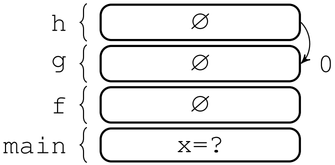


. . . Finally `g` calls `h()`

Repetition (continued)

```c
int h(void) {
  return 0;
}
int g(void) {
  return 1+h();
}
int f(void) {
  return 1+g();
}
int main(void) {
  int x = 1+f();
  return 0;
}
```

Call Stack:

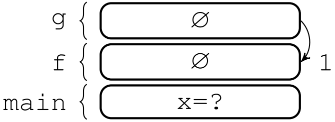


. . . Then `h` returns `0` to the calling function `g`

Repetition (continued)

```c
int h(void) {
  return 0;
}
int g(void) {
  return 1+h();
}
int f(void) {
  return 1+g();
}
int main(void) {
  int x = 1+f();
  return 0;
}
```

Call Stack:

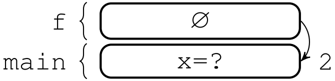


. . . Then `g` returns `1+0=1` to `f`

Repetition (continued)

```
int h(void) {
  return 0;
}
int g(void) {
  return 1+h();
}
int f(void) {
  return 1+g();
}
int main(void) {
  int x = 1+f();
  return 0;
}
```

Call Stack:



. . . Then `f` returns `1+1=2` to `main`

Repetition (continued)

```c
int h(void) {
  return 0;
}
int g(void) {
  return 1+h();
}
int f(void) {
  return 1+g();
}
int main(void) {
  int x = 1+f();
  return 0;
}
```

Call Stack:

main { x=3 }

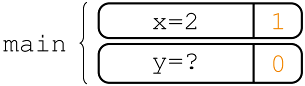
. . . Finally `main` gets the control again and `x=1+2=3`.

Repetition (continued)

Parameters, Local variables and Stack: an example.

```
int foo(int a) {
  int b = 0;
  b = 3+a;
  a = 10;
  return b;
}
int main(void) {
  int x = 2;
  int y = foo(x);
  return 0;
}
```

Call Stack:



When a function is called (even `main`) its <u>all</u> variables are pushed into the stack!

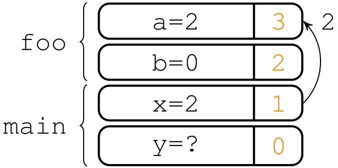
Note: numbers between brackets denote a position in the stack but, in real, they are memory addresses.

Repetition (continued)

Parameters, Local variables and Stack: an example.

```c
int foo(int a) {
  int b = 0;
  b = 3+a;
  a = 10;
  return b;
}
int main(void) {
  int x = 2;
  int y = foo(x);
  return 0;
}
```

Call Stack:



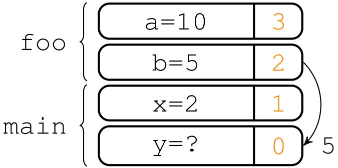
`foo` is called and `a` gets the value of `x`!

Repetition (continued)

Parameters, Local variables and Stack: an example.

```
int foo(int a) {
  int b = 0;
  b = 3+a;
  a = 10;
  return b;
}
int main(void) {
  int x = 2;
  int y = foo(x);
  return 0;
}
```

Call Stack:



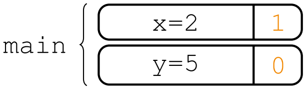
`foo` ends, `a` and `b` are changed and `b` is returned to `main`.

Repetition (continued)

Parameters, Local variables and Stack: an example.

```c
int foo(int a) {
  int b = 0;
  b = 3+a;
  a = 10;
  return b;
}
int main(void) {
  int x = 2;
  int y = foo(x);
  return 0;
}
```

Call Stack:



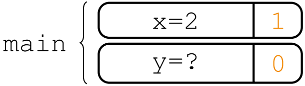
`foo` has been removed from the stack and `y` gets `5`.

Repetition (continued)

Pointer Parameters, Local variables and Stack: an example.

```c
int foo(int * a) {
  int b = 0;
  b = 3 + *a;
  *a = 10;
  return b;
}
int main(void) {
  int x = 2;
  int y = foo(&x);
  return 0;
}
```

Call Stack:

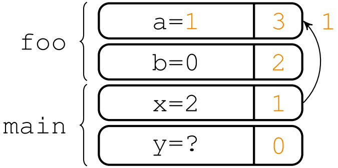


Repetition (continued)

Pointer Parameters, Local variables and Stack: an example.

```
int foo(int * a) {
  int b = 0;
  b = 3 + *a;
  *a = 10;
  return b;
}
int main(void) {
  int x = 2;
  int y = foo(&x);
  return 0;
}
```

Call Stack:



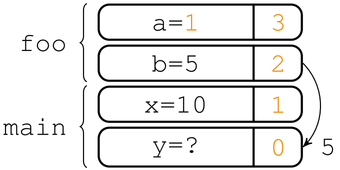
Here the parameter `a` is a pointer and it gets `&x` which is the position of `x` in the stack (i.e., its memory address).

Repetition (continued)

Pointer Parameters, Local variables and Stack: an example.

```
int foo(int * a) {
  int b = 0;
  b = 3 + *a;
  *a = 10;
  return b;
}
int main(void) {
  int x = 2;
  int y = foo(&x);
  return 0;
}
```

Call Stack:



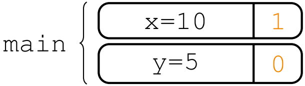
Since `a` points to `x`, to update `*a` (i.e., the value pointed by `a`) means to update `x`, while the value of `a` does **not** change.

Repetition (continued)

Pointer Parameters, Local variables and Stack: an example.

```
int foo(int * a) {
  int b = 0;
  b = 3 + *a;
  *a = 10;
  return b;
}
int main(void) {
  int x = 2;
  int y = foo(&x);
  return 0;
}
```

Call Stack:



When `foo` returns, `foo` and its variables are removed from the stack and the control returns to `main` with `x` changed.

Recursion

A **recursive function** is a function which calls itself in the definition.

Each recursive function must have at least one **base case**, as well as the **recursive case**:

- Base case: the case for which the solution can be stated non-recursively.
- Recursive case: the case for which the solution is expressed in terms of a smaller version of itself.

Note that it can happen that a chain of function-calls returns to the method that originated the chain, in this case we have **indirect recursion**.

Recursive functions are important because they allow to define easily and in a more elegant way algorithms (e.g., sorting functions) and data-structures (e.g., linked lists and trees).

Recursion - example

The factorial function can be defined as:

$$n! = \begin{cases} \prod_{i=1}^{n} i & \text{if } n > 1, \\ 1 & \text{otherwise} \end{cases}$$

...Which is equivalent to:

$$n! = \begin{cases} n \cdot (n-1)! & \text{if } n > 1, \\ 1 & \text{otherwise} \end{cases}$$

For example $5! = 5 \cdot \underbrace{4 \cdot \underbrace{3 \cdot \underbrace{2 \cdot 1}_{2!}}_{3!}}_{4!} = 120$

How can we implement these two definitions?

Recursion - example (continued)

Iterative implementation:

```
int f(int n) {
  int result = 1;
  while(n>1) {
    result = result*n;
    n = n-1;
  }
  return result;
}
```

Recursive implementation:

```
int f(int n) {
  if(n>1)
    return n*f(n-1); //recursive case
  else
    return 1; //base case
}
```

Recursion - example (continued)

Recursion and Stack: an example.

```
int f(int n) {
  if(n>1)
    return n*f(n-1);
  else
    return 1;
}

int main(void) {
  int x = f(3);
  return 0;
}
```

Call Stack:

main { x=?

Recursion - example (continued)

Recursion and Stack: an example.

```
int f(int n) {
  if(n>1)
    return n*f(n-1);
  else
    return 1;
}

int main(void) {
  int x = f(3);
  return 0;
}
```

Call Stack:

f	n=3
main	x=?

Until the formal parameter is greater than `1`, the recursive case of `f` is chosen and, every time, the program pushes on the stack a new instance of `f` by passing a value of `n` decreased.

Recursion - example (continued)

Recursion and Stack: an example.

```
int f(int n) {
  if(n>1)
    return n*f(n-1);
  else
    return 1;
}

int main(void) {
  int x = f(3);
  return 0;
}
```

Call Stack:

```
f    { n=2
f    { n=3
main { x=?
```

Until the formal parameter is greater than `1`, the recursive case of `f` is chosen and, every time, the program pushes on the stack a new instance of `f` by passing a value of `n` decreased.

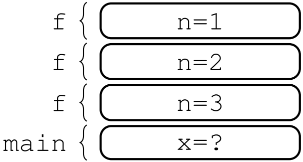
Recursion - example (continued)

Recursion and Stack: an example.

```c
int f(int n) {
  if(n>1)
    return n*f(n-1);
  else
    return 1;
}

int main(void) {
  int x = f(3);
  return 0;
}
```

Call Stack:



Until the formal parameter is greater than `1`, the recursive case of `f` is chosen and, every time, the program pushes on the stack a new instance of `f` by passing a value of `n` decreased.

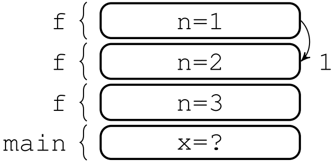
Recursion - example (continued)

Recursion and Stack: an example.

```
int f(int n) {
  if(n>1)
    return n*f(n-1);
  else
    return 1;
}

int main(void) {
  int x = f(3);
  return 0;
}
```

Call Stack:



When the formal parameter equals `1`, the base case of `f` is chosen and `1` is returned to the previous instance of `f` in the stack.

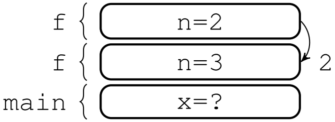
Recursion - example (continued)

Recursion and Stack: an example.

```
int f(int n) {
  if(n>1)
    return n*f(n-1);
  else
    return 1;
}

int main(void) {
  int x = f(3);
  return 0;
}
```

Call Stack:



One by one, the instances of `f`, that followed the general case, restart and return their value of `n` (which has never changed) multiplied by the value returned by `f` that they called.

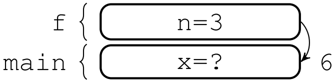
Recursion - example (continued)

Recursion and Stack: an example.

```
int f(int n) {
  if(n>1)
    return n*f(n-1);
  else
    return 1;
}

int main(void) {
  int x = f(3);
  return 0;
}
```

Call Stack:



Recursion - example (continued)

Recursion and Stack: an example.

```
int f(int n) {
  if(n>1)
    return n*f(n-1);
  else
    return 1;
}

int main(void) {
  int x = f(3);
  return 0;
}
```

Call Stack:

main { x=6

Recursion vs Iterations

- Iterations (or Loops)
  - have at least one termination (e.g., `while(...)`),
  - something has to change in each turn so that the condition is reached (e.g., `n = n-1`).
- Recursion
  - has at least one base case (e.g., `if(...)  return 0;`),
  - the argument must change in each recursive call so that the base case occurs (e.g., `f(n-1)`).

Loops and recursion are computationally equivalent, that is everything can be done with a loop can be also done with recursion and vice-versa. However, it may be some differences in practice/effectiveness.

Recursion - test

Consider the following function:

```
int f(int i, int x) {
    if(i>0 && x<=15)
        return 2+f(i-1, x*2);
    else
        return i;
}
```

What is the result from the call `f(2,5)` *and* `f(5,1)`*?*

Dynamic memory, Heap & Pointers

- Stack (repetita iuvant)
  - When a function is called, a block is reserved on the top of the stack for local variables (and something more). When the function returns, the block becomes unused and, then, it's freed.
  - Memory blocks is reserved in a LIFO order: the most recently reserved block is always the next block to be freed.
  - It is "automagically" managed by the system.
- **Heap**
  - It is for dynamic allocation.
  - Unlike the stack, there's no specific pattern for the allocation of blocks.
  - It is managed by the programmer: `malloc()`, `calloc()`, `realloc()`, `free()`, ...
  - Memory is byte addressed. Each variable type has a specific size which can be get by means of the function `sizeof()`.

Pointers - test

Consider the following program:

```
void foo(int *A, int n) {
  int i = 0;
  int s = sizeof(int);
  A = (int*)malloc(n*s);
  while(i<n)
    A[i++] = 10;
}

int main(void) {
  int *x = NULL;
  foo(x,3);
  printf("%d", x[0]);
}
```

What does it print?

Pointers - test (continued)

```
void foo(int *A, int n) {
  int i = 0;
  int s = sizeof(int);
  A = (int*)malloc(n*s);
  while(i<n)
    A[i++] = 10;
}

int main(void) {
  int *x = NULL;
  foo(x,3);
  printf("%d", x[0]);
}
```

Call Stack:

`main` { x=NULL

Heap:

$\varnothing$

The computation starts from the `main` function. . .

Pointers - test (continued)

```c
void foo(int *A, int n) {
  int i = 0;
  int s = sizeof(int);
  A = (int*)malloc(n*s);
  while(i<n)
    A[i++] = 10;
}

int main(void) {
  int *x = NULL;
  foo(x,3);
  printf("%d", x[0]);
}
```

Call Stack:

foo	s=4
	i=0
	n=3
	A=?
main	x=NULL

Heap:

$\emptyset$

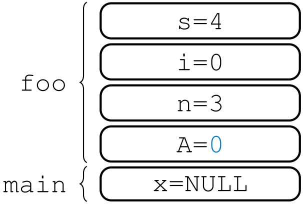
. . . The `main` function calls `foo`. Here `s` equals `4` since we need 4 bytes to store an `int`. . .

Pointers - test (continued)

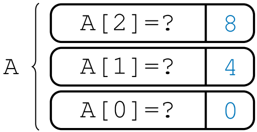
```
void foo(int *A, int n) {
  int i = 0;
  int s = sizeof(int);
  A = (int*)malloc(n*s);
  while(i<n)
    A[i++] = 10;
}

int main(void) {
  int *x = NULL;
  foo(x,3);
  printf("%d", x[0]);
}
```

Call Stack:



Heap:



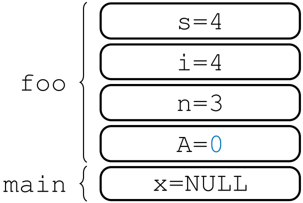
. . . The `malloc` function allocates 12 bytes in the heap. . .

Pointers - test (continued)

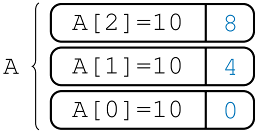
```c
void foo(int *A, int n) {
  int i = 0;
  int s = sizeof(int);
  A = (int*)malloc(n*s);
  while(i<n)
    A[i++] = 10;
}

int main(void) {
  int *x = NULL;
  foo(x,3);
  printf("%d", x[0]);
}
```

Call Stack:



Heap:



. . . Then the array `A` is initialized. . .

Pointers - test (continued)

```
void foo(int *A, int n) {
  int i = 0;
  int s = sizeof(int);
  A = (int*)malloc(n*s);
  while(i<n)
    A[i++] = 10;
}

int main(void) {
  int *x = NULL;
  foo(x,3);
  printf("%d", x[0]);
}
```

Call Stack:

main { x=NULL

Heap:

A	A[2]=10	8
	A[1]=10	4
	A[0]=10	0

. . . The function `foo` ends and is removed from the stack: `A` is still alive in the heap, but `x` is still `NULL`. When `main` accesses `x[0]`, we get a `Segmentation fault` error!

Pointers - Solution #1

```
int* foo(int n) {
  int i = 0;
  int s = sizeof(int);
  int *A = (int*)malloc(n*s);
  while(i<n)
    A[i++] = 10;
  return A;
}

int main(void) {
  int *x = NULL;
  x = foo(3);
  printf("%d", x[0]);
}
```

Double Pointers

Pointers are variables, like `int`, so that they are stored somewhere in memory. As a consequence, we can handle their memory address as we usually do with `int`:

```c
int main(void) {
  int x = 5;
  int * px = &x;
  int ** ppx = &px;
  //we could continue with triple pointers and so on
}
```

ppx=1	2
px=0	1
x=5	0

Pointers - Solution #2

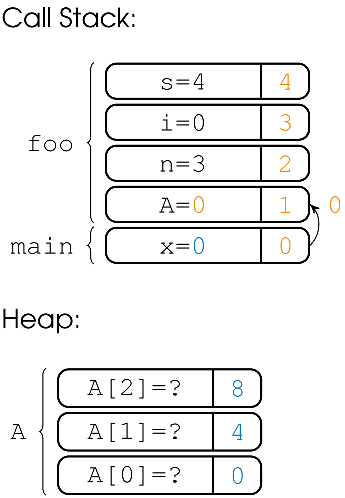
```
void foo(int **A, int n) {
  int i = 0;
  int s = sizeof(int);
  *A = (int*)malloc(n*s);
  while(i<n)
    (*A)[i++] = 10;
}

int main(void) {
  int *x = NULL;
  foo(&x,3);
  printf("%d", x[0]);
}
```

Pointers - Solution #2 (continued)

```c
void foo(int **A, int n) {
  int i = 0;
  int s = sizeof(int);
  *A = (int*)malloc(n*s);
  while(i<n)
    (*A)[i++] = 10;
}

int main(void) {
  int *x = NULL;
  foo(&x,3);
  printf("%d", x[0]);
}
```



...The `malloc` function still allocates 12 bytes in the heap, but this time the target variable is `x` (which is denoted by `*A`).

Pointers

Do we forget something?

Pointers

Do we forget something? Yes, we didn't dispose the memory!

Heap management is your responsibility. Heap is not boundless: if you don't `free` the next `malloc` could fail!

```
int main(void) {
  int *x = NULL;
  foo(&x,3);
  printf("%d", x[0]);
  free(x);
}
```