

# Recursion – Multi-dimensional arrays

## Basics of Programming 1



G. Horváth, A.B. Nagy, Z. Zsóka, P. Fiala, A. Vitéz

18 November, 2018

## 1 Recursion

- Definition
- Writing recursive programs
- Recursion or iteration
- Applications
- Indirect recursion

## 2 Multi-dimensional arrays

- Definition
- Passing as argument to function
- Dynamic 2D array
- Array of pointers

# Chapter 1

## Recursion

# Recursion – definition

Many mathematical problems can be formulated recursively

# Recursion – definition

Many mathematical problems can be formulated recursively

- Sum of sequence  $a_n$

$$S_n = \begin{cases} S_{n-1} + a_n & n > 0 \\ a_0 & n = 0 \end{cases}$$

# Recursion – definition

Many mathematical problems can be formulated recursively

- Sum of sequence  $a_n$

$$S_n = \begin{cases} S_{n-1} + a_n & n > 0 \\ a_0 & n = 0 \end{cases}$$

- Factorial

$$n! = \begin{cases} (n-1)! \cdot n & n > 0 \\ 1 & n = 0 \end{cases}$$

# Recursion – definition

Many mathematical problems can be formulated recursively

- Sum of sequence  $a_n$

$$S_n = \begin{cases} S_{n-1} + a_n & n > 0 \\ a_0 & n = 0 \end{cases}$$

- Factorial

$$n! = \begin{cases} (n-1)! \cdot n & n > 0 \\ 1 & n = 0 \end{cases}$$

- Fibonacci numbers

$$F_n = \begin{cases} F_{n-2} + F_{n-1} & n > 1 \\ 1 & n = 1 \\ 0 & n = 0 \end{cases}$$

# Recursion – definition

Several everyday problems can be formulated recursively



# Recursion – definition

Several everyday problems can be formulated recursively

- Is Albert Einstein my ancestor?

# Recursion – definition

Several everyday problems can be formulated recursively

- Is Albert Einstein my ancestor?

$$\text{My ancestor?} = \left\{ \begin{array}{l} \text{Ancestor of my father/mother?} \\ \end{array} \right.$$

# Recursion – definition

Several everyday problems can be formulated recursively

- Is Albert Einstein my ancestor?

$$\text{My ancestor?} = \begin{cases} \text{Ancestor of my father/mother?} \\ \text{Is he my father?} \end{cases}$$

# Recursion – definition

Several everyday problems can be formulated recursively

- Is Albert Einstein my ancestor?

$$\text{My ancestor?} = \begin{cases} \text{Ancestor of my father/mother?} \\ \text{Is he my father?} \\ \text{Is she my mother?} \end{cases}$$

# Recursion – definition

Several everyday problems can be formulated recursively

- Is Albert Einstein my ancestor?

$$\text{My ancestor?} = \begin{cases} \text{Ancestor of my father/mother?} \\ \text{Is he my father?} \\ \text{Is she my mother?} \end{cases}$$

- In general

$$\text{Problem} = \begin{cases} \text{Simpler, similar problem(s)} \\ \text{Trivial case(es)} \end{cases}$$

# Recursion – outlook

- Recursion is useful in many areas

# Recursion – outlook

- Recursion is useful in many areas  
*Mathematical proof* e.g., proof by induction

# Recursion – outlook

- Recursion is useful in many areas
  - Mathematical proof e.g., proof by induction
  - Definition e.g., Fibonacci numbers



# Recursion – outlook

- Recursion is useful in many areas
  - Mathematical proof e.g., proof by induction
    - Definition e.g., Fibonacci numbers
    - Algorithm e.g., path finding algorithms

# Recursion – outlook

- Recursion is useful in many areas
  - Mathematical proof e.g., proof by induction
    - Definition e.g., Fibonacci numbers
    - Algorithm e.g., path finding algorithms
  - Data structure e.g., linked list, folders of the op. system

# Recursion – outlook

- Recursion is useful in many areas
  - Mathematical proof e.g., proof by induction
    - Definition e.g., Fibonacci numbers
    - Algorithm e.g., path finding algorithms
  - Data structure e.g., linked list, folders of the op. system
  - Geometric constructions e.g., fractals

# Recursion – outlook

- Recursion is useful in many areas
  - Mathematical proof e.g., proof by induction
    - Definition e.g., Fibonacci numbers
    - Algorithm e.g., path finding algorithms
  - Data structure e.g., linked list, folders of the op. system
  - Geometric constructions e.g., fractals
- We are going to study recursive data structures and recursive algorithms

# Recursive algorithms in C

## ■ Factorial

$$n! = \begin{cases} (n-1)! \cdot n & n > 0 \\ 1 & n = 0 \end{cases}$$

$$5! = 4! \cdot 5$$

# Recursive algorithms in C

## ■ Factorial

$$n! = \begin{cases} (n-1)! \cdot n & n > 0 \\ 1 & n = 0 \end{cases}$$

$$5! = 4! \cdot 5$$

# Recursive algorithms in C

## ■ Factorial

$$n! = \begin{cases} (n-1)! \cdot n & n > 0 \\ 1 & n = 0 \end{cases}$$

$$5! = (3! \cdot 4) \cdot 5$$

# Recursive algorithms in C

## ■ Factorial

$$n! = \begin{cases} (n-1)! \cdot n & n > 0 \\ 1 & n = 0 \end{cases}$$

$$5! = (3! \cdot 4) \cdot 5$$



# Recursive algorithms in C

## ■ Factorial

$$n! = \begin{cases} (n-1)! \cdot n & n > 0 \\ 1 & n = 0 \end{cases}$$

$$5! = ((2! \cdot 3) \cdot 4) \cdot 5$$

# Recursive algorithms in C

## ■ Factorial

$$n! = \begin{cases} (n-1)! \cdot n & n > 0 \\ 1 & n = 0 \end{cases}$$

$$5! = ((2! \cdot 3) \cdot 4) \cdot 5$$

# Recursive algorithms in C

## ■ Factorial

$$n! = \begin{cases} (n-1)! \cdot n & n > 0 \\ 1 & n = 0 \end{cases}$$

$$5! = (((1! \cdot 2) \cdot 3) \cdot 4) \cdot 5$$

# Recursive algorithms in C

## ■ Factorial

$$n! = \begin{cases} (n-1)! \cdot n & n > 0 \\ 1 & n = 0 \end{cases}$$

$$5! = (((1! \cdot 2) \cdot 3) \cdot 4) \cdot 5$$

# Recursive algorithms in C

## ■ Factorial

$$n! = \begin{cases} (n-1)! \cdot n & n > 0 \\ 1 & n = 0 \end{cases}$$

$$5! = (((((0! \cdot 1) \cdot 2) \cdot 3) \cdot 4) \cdot 5$$

# Recursive algorithms in C

## ■ Factorial

$$n! = \begin{cases} (n-1)! \cdot n & n > 0 \\ 1 & n = 0 \end{cases}$$

$$5! = (((((0! \cdot 1) \cdot 2) \cdot 3) \cdot 4) \cdot 5$$

# Recursive algorithms in C

## ■ Factorial

$$n! = \begin{cases} (n-1)! \cdot n & n > 0 \\ 1 & n = 0 \end{cases}$$

$$5! = (((((1 \cdot 1) \cdot 2) \cdot 3) \cdot 4) \cdot 5)$$

# Recursive algorithms in C

## ■ Factorial

$$n! = \begin{cases} (n-1)! \cdot n & n > 0 \\ 1 & n = 0 \end{cases}$$

$$5! = (((((1 \cdot 1) \cdot 2) \cdot 3) \cdot 4) \cdot 5$$



# Recursive algorithms in C

## ■ Factorial

$$n! = \begin{cases} (n-1)! \cdot n & n > 0 \\ 1 & n = 0 \end{cases}$$

$$5! = (((1 \cdot 2) \cdot 3) \cdot 4) \cdot 5$$

# Recursive algorithms in C

## ■ Factorial

$$n! = \begin{cases} (n-1)! \cdot n & n > 0 \\ 1 & n = 0 \end{cases}$$

$$5! = (((1 \cdot 2) \cdot 3) \cdot 4) \cdot 5$$

# Recursive algorithms in C

## ■ Factorial

$$n! = \begin{cases} (n-1)! \cdot n & n > 0 \\ 1 & n = 0 \end{cases}$$

$$5! = ((2 \cdot 3) \cdot 4) \cdot 5$$

# Recursive algorithms in C

## ■ Factorial

$$n! = \begin{cases} (n-1)! \cdot n & n > 0 \\ 1 & n = 0 \end{cases}$$

$$5! = ((2 \cdot 3) \cdot 4) \cdot 5$$

# Recursive algorithms in C

## ■ Factorial

$$n! = \begin{cases} (n-1)! \cdot n & n > 0 \\ 1 & n = 0 \end{cases}$$

$$5! = (6 \cdot 4) \cdot 5$$

# Recursive algorithms in C

## ■ Factorial

$$n! = \begin{cases} (n-1)! \cdot n & n > 0 \\ 1 & n = 0 \end{cases}$$

$$5! = (6 \cdot 4) \cdot 5$$

# Recursive algorithms in C

## ■ Factorial

$$n! = \begin{cases} (n-1)! \cdot n & n > 0 \\ 1 & n = 0 \end{cases}$$

$$5! = 24 \cdot 5$$

# Recursive algorithms in C

## ■ Factorial

$$n! = \begin{cases} (n-1)! \cdot n & n > 0 \\ 1 & n = 0 \end{cases}$$

$$5! = 24 \cdot 5$$



# Recursive algorithms in C

## ■ Factorial

$$n! = \begin{cases} (n-1)! \cdot n & n > 0 \\ 1 & n = 0 \end{cases}$$

$$5! = 120$$

# Recursive algorithms in C

## ■ Factorial

$$n! = \begin{cases} (n-1)! \cdot n & n > 0 \\ 1 & n = 0 \end{cases}$$

# Recursive algorithms in C

## ■ Factorial

$$n! = \begin{cases} (n-1)! \cdot n & n > 0 \\ 1 & n = 0 \end{cases}$$

Let us implement it to C!

```
1 unsigned factorial(unsigned n)
2 {
3     if (n > 0)
4         return factorial(n-1) * n;
5     else
6         return 1;
7 }
```

# Recursive algorithms in C

## ■ Factorial

$$n! = \begin{cases} (n-1)! \cdot n & n > 0 \\ 1 & n = 0 \end{cases}$$

Let us implement it to C!

```
1 unsigned factorial(unsigned n)
2 {
3     if (n > 0)
4         return factorial(n-1) * n;
5     else
6         return 1;
7 }
```

## ■ Calling the function

```
1 unsigned f = factorial(5); /* it works! */
2 printf("%u\n", f);
```

# Some considerations

## ■ How to imagine recursive functions?

```
1 unsigned f0(void) { return 1; }  
2 unsigned f1(void) { return f0() * 1; }  
3 unsigned f2(void) { return f1() * 2; }  
4 unsigned f3(void) { return f2() * 3; }  
5 unsigned f4(void) { return f3() * 4; }  
6 unsigned f5(void) { return f4() * 5; }  
7 ...  
8 unsigned f = f5();
```

- Many different instances of the same function coexist simultaneously
- The instances were called with different parameters

# Implementing recursion

How can multiple instances of the same function coexist?

# Implementing recursion

How can multiple instances of the same function coexist?

```
1  /*
2   recursive factorial function
3  */
4  unsigned factorial(unsigned n)
5  {
6      if (n > 0)
7          return factorial(n-1) * n;
8      else
9          return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

register:

# Implementing recursion

How can multiple instances of the same function coexist?

```
1  /*
2   recursive factorial function
3  */
4  unsigned factorial(unsigned n)
5  {
6      if (n > 0)
7          return factorial(n-1) * n;
8      else
9          return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

register:



# Implementing recursion

How can multiple instances of the same function coexist?

```
1  /*
2   recursive factorial function
3   */
4  unsigned factorial(unsigned n)
5  {
6     if (n > 0)
7         return factorial(n-1) * n;
8     else
9         return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x2000:

4

register:

??

# Implementing recursion

How can multiple instances of the same function coexist?

```
1  /*
2   recursive factorial function
3  */
4  unsigned factorial(unsigned n)
5  {
6      if (n > 0)
7          return factorial(n-1) * n;
8      else
9          return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FFC:	15
0x2000:	4

register:	??
-----------	----

# Implementing recursion

How can multiple instances of the same function coexist?

```
1  /*
2   recursive factorial function
3  */
4  unsigned factorial(unsigned n)
5  {
6      if (n > 0)
7          return factorial(n-1) * n;
8      else
9          return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FFC:	15
n 0x2000:	4
register:	??

# Implementing recursion

How can multiple instances of the same function coexist?

```
1  /*
2   recursive factorial function
3  */
4  unsigned factorial(unsigned n)
5  {
6     if (n > 0)
7         return factorial(n-1) * n;
8     else
9         return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FFC:	15
n 0x2000:	4

register:	??
-----------	----

# Implementing recursion

How can multiple instances of the same function coexist?

```
1  /*
2   recursive factorial function
3  */
4  unsigned factorial(unsigned n)
5  {
6     if (n > 0)
7         return factorial(n-1) * n;
8     else
9         return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FFC:	15
n 0x2000:	4

register:	??
-----------	----

# Implementing recursion

How can multiple instances of the same function coexist?

```

1  /*
2   recursive factorial function
3  */
4  unsigned factorial(unsigned n)
5  {
6      if (n > 0)
7          return factorial(n-1) * n;
8      else
9          return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FF8:	3
0x1FFC:	15
n 0x2000:	4
register:	??

# Implementing recursion

How can multiple instances of the same function coexist?

```

1  /*
2   recursive factorial function
3  */
4  unsigned factorial(unsigned n)
5  {
6   if (n > 0)
7   return factorial(n-1) * n;
8   else
9   return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FF4:	7
0x1FF8:	3
0x1FFC:	15
n 0x2000:	4

register: ??

# Implementing recursion

How can multiple instances of the same function coexist?

```

1  /*
2   recursive factorial function
3  */
4  unsigned factorial(unsigned n)
5  {
6      if (n > 0)
7          return factorial(n-1) * n;
8      else
9          return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FF4:	7
n 0x1FF8:	3
0x1FFC:	15
0x2000:	4
register:	??



# Implementing recursion

How can multiple instances of the same function coexist?

```

1  /*
2   recursive factorial function
3  */
4  unsigned factorial(unsigned n)
5  {
6     if (n > 0)
7         return factorial(n-1) * n;
8     else
9         return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FF4:	7
n 0x1FF8:	3
0x1FFC:	15
0x2000:	4
register:	??

# Implementing recursion

How can multiple instances of the same function coexist?

```
1  /*
2   recursive factorial function
3  */
4  unsigned factorial(unsigned n)
5  {
6   if (n > 0)
7   return factorial(n-1) * n;
8   else
9   return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FF4:	7
n 0x1FF8:	3
0x1FFC:	15
0x2000:	4
register:	??

# Implementing recursion

How can multiple instances of the same function coexist?

```

1  /*
2   recursive factorial function
3  */
4  unsigned factorial(unsigned n)
5  {
6   if (n > 0)
7   return factorial(n-1) * n;
8   else
9   return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FF0:	2
0x1FF4:	7
n 0x1FF8:	3
0x1FFC:	15
0x2000:	4
register:	??

# Implementing recursion

How can multiple instances of the same function coexist?

```

1  /*
2   recursive factorial function
3   */
4  unsigned factorial(unsigned n)
5  {
6   if (n > 0)
7     return factorial(n-1) * n;
8   else
9     return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FEC:	7
0x1FF0:	2
0x1FF4:	7
n 0x1FF8:	3
0x1FFC:	15
0x2000:	4
register:	??

# Implementing recursion

How can multiple instances of the same function coexist?

```

1  /*
2   recursive factorial function
3   */
4  unsigned factorial(unsigned n)
5  {
6      if (n > 0)
7          return factorial(n-1) * n;
8      else
9          return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FEC:	7
n 0x1FF0:	2
0x1FF4:	7
0x1FF8:	3
0x1FFC:	15
0x2000:	4

register: ??

# Implementing recursion

How can multiple instances of the same function coexist?

```

1  /*
2   recursive factorial function
3  */
4  unsigned factorial(unsigned n)
5  {
6     if (n > 0)
7         return factorial(n-1) * n;
8     else
9         return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FEC:	7
n 0x1FF0:	2
0x1FF4:	7
0x1FF8:	3
0x1FFC:	15
0x2000:	4

register: ??

# Implementing recursion

How can multiple instances of the same function coexist?

```

1  /*
2   recursive factorial function
3  */
4  unsigned factorial(unsigned n)
5  {
6      if (n > 0)
7          return factorial(n-1) * n;
8      else
9          return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FEC:	7
n 0x1FF0:	2
0x1FF4:	7
0x1FF8:	3
0x1FFC:	15
0x2000:	4

register: ??

# Implementing recursion

How can multiple instances of the same function coexist?

```

1  /*
2   recursive factorial function
3  */
4  unsigned factorial(unsigned n)
5  {
6      if (n > 0)
7          return factorial(n-1) * n;
8      else
9          return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FE8:	1
0x1FEC:	7
n 0x1FF0:	2
0x1FF4:	7
0x1FF8:	3
0x1FFC:	15
0x2000:	4

register: ??



# Implementing recursion

How can multiple instances of the same function coexist?

```

1  /*
2   recursive factorial function
3   */
4  unsigned factorial(unsigned n)
5  {
6     if (n > 0)
7         return factorial(n-1) * n;
8     else
9         return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FE4:	7
0x1FE8:	1
0x1FEC:	7
n 0x1FF0:	2
0x1FF4:	7
0x1FF8:	3
0x1FFC:	15
0x2000:	4

register: ??

# Implementing recursion

How can multiple instances of the same function coexist?

```

1  /*
2   recursive factorial function
3   */
4  unsigned factorial(unsigned n)
5  {
6      if (n > 0)
7          return factorial(n-1) * n;
8      else
9          return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FE4:	7
n 0x1FE8:	1
0x1FEC:	7
0x1FF0:	2
0x1FF4:	7
0x1FF8:	3
0x1FFC:	15
0x2000:	4

register: ??

# Implementing recursion

How can multiple instances of the same function coexist?

```

1  /*
2   recursive factorial function
3   */
4  unsigned factorial(unsigned n)
5  {
6     if (n > 0)
7         return factorial(n-1) * n;
8     else
9         return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FE4:	7
n 0x1FE8:	1
0x1FEC:	7
0x1FF0:	2
0x1FF4:	7
0x1FF8:	3
0x1FFC:	15
0x2000:	4

register: ??

# Implementing recursion

How can multiple instances of the same function coexist?

```

1  /*
2   recursive factorial function
3   */
4  unsigned factorial(unsigned n)
5  {
6      if (n > 0)
7          return factorial(n-1) * n;
8      else
9          return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FE4:	7
n 0x1FE8:	1
0x1FEC:	7
0x1FF0:	2
0x1FF4:	7
0x1FF8:	3
0x1FFC:	15
0x2000:	4

register: ??

# Implementing recursion

How can multiple instances of the same function coexist?

```

1  /*
2   recursive factorial function
3  */
4  unsigned factorial(unsigned n)
5  {
6      if (n > 0)
7          return factorial(n-1) * n;
8      else
9          return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FE0:	0
0x1FE4:	7
n 0x1FE8:	1
0x1FEC:	7
0x1FF0:	2
0x1FF4:	7
0x1FF8:	3
0x1FFC:	15
0x2000:	4

register: ??

# Implementing recursion

How can multiple instances of the same function coexist?

```

1  /*
2   recursive factorial function
3  */
4  unsigned factorial(unsigned n)
5  {
6      if (n > 0)
7          return factorial(n-1) * n;
8      else
9          return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FDC:	7
0x1FE0:	0
0x1FE4:	7
n 0x1FE8:	1
0x1FEC:	7
0x1FF0:	2
0x1FF4:	7
0x1FF8:	3
0x1FFC:	15
0x2000:	4

register: ??

# Implementing recursion

How can multiple instances of the same function coexist?

```

1  /*
2   recursive factorial function
3  */
4  unsigned factorial(unsigned n)
5  {
6      if (n > 0)
7          return factorial(n-1) * n;
8      else
9          return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FDC:	7
n 0x1FE0:	0
0x1FE4:	7
0x1FE8:	1
0x1FEC:	7
0x1FF0:	2
0x1FF4:	7
0x1FF8:	3
0x1FFC:	15
0x2000:	4

register: ??

# Implementing recursion

How can multiple instances of the same function coexist?

```

1  /*
2   recursive factorial function
3  */
4  unsigned factorial(unsigned n)
5  {
6     if (n > 0)
7         return factorial(n-1) * n;
8     else
9         return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FDC:	7
n 0x1FE0:	0
0x1FE4:	7
0x1FE8:	1
0x1FEC:	7
0x1FF0:	2
0x1FF4:	7
0x1FF8:	3
0x1FFC:	15
0x2000:	4

register: ??



# Implementing recursion

How can multiple instances of the same function coexist?

```

1  /*
2   recursive factorial function
3  */
4  unsigned factorial(unsigned n)
5  {
6      if (n > 0)
7          return factorial(n-1) * n;
8      else
9          return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FDC:	7
n 0x1FE0:	0
0x1FE4:	7
0x1FE8:	1
0x1FEC:	7
0x1FF0:	2
0x1FF4:	7
0x1FF8:	3
0x1FFC:	15
0x2000:	4

register: ??

# Implementing recursion

How can multiple instances of the same function coexist?

```

1  /*
2   recursive factorial function
3  */
4  unsigned factorial(unsigned n)
5  {
6      if (n > 0)
7          return factorial(n-1) * n;
8      else
9          return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FDC:	7
n 0x1FE0:	0
0x1FE4:	7
0x1FE8:	1
0x1FEC:	7
0x1FF0:	2
0x1FF4:	7
0x1FF8:	3
0x1FFC:	15
0x2000:	4

register: 1

# Implementing recursion

How can multiple instances of the same function coexist?

```

1  /*
2   recursive factorial function
3  */
4  unsigned factorial(unsigned n)
5  {
6      if (n > 0)
7          return factorial(n-1) * n;
8      else
9          return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FDC:	7
n 0x1FE0:	0
0x1FE4:	7
0x1FE8:	1
0x1FEC:	7
0x1FF0:	2
0x1FF4:	7
0x1FF8:	3
0x1FFC:	15
0x2000:	4

register: 1

# Implementing recursion

How can multiple instances of the same function coexist?

```

1  /*
2   recursive factorial function
3   */
4  unsigned factorial(unsigned n)
5  {
6      if (n > 0)
7          return factorial(n-1) * n;
8      else
9          return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FE4:	7
n 0x1FE8:	1
0x1FEC:	7
0x1FF0:	2
0x1FF4:	7
0x1FF8:	3
0x1FFC:	15
0x2000:	4

register: 1

# Implementing recursion

How can multiple instances of the same function coexist?

```

1  /*
2   recursive factorial function
3   */
4  unsigned factorial(unsigned n)
5  {
6      if (n > 0)
7          return factorial(n-1) * n;
8      else
9          return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FE4:	7
n 0x1FE8:	1
0x1FEC:	7
0x1FF0:	2
0x1FF4:	7
0x1FF8:	3
0x1FFC:	15
0x2000:	4

register: 1

# Implementing recursion

How can multiple instances of the same function coexist?

```

1  /*
2   recursive factorial function
3   */
4  unsigned factorial(unsigned n)
5  {
6      if (n > 0)
7          return factorial(n-1) * n;
8      else
9          return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FE4:	7
n 0x1FE8:	1
0x1FEC:	7
0x1FF0:	2
0x1FF4:	7
0x1FF8:	3
0x1FFC:	15
0x2000:	4

register:	1
-----------	---

# Implementing recursion

How can multiple instances of the same function coexist?

```

1  /*
2   recursive factorial function
3   */
4  unsigned factorial(unsigned n)
5  {
6      if (n > 0)
7          return factorial(n-1) * n;
8      else
9          return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FEC:	7
n 0x1FF0:	2
0x1FF4:	7
0x1FF8:	3
0x1FFC:	15
0x2000:	4

register:	1
-----------	---

# Implementing recursion

How can multiple instances of the same function coexist?

```

1  /*
2   recursive factorial function
3  */
4  unsigned factorial(unsigned n)
5  {
6      if (n > 0)
7          return factorial(n-1) * n;
8      else
9          return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FEC:	7
n 0x1FF0:	2
0x1FF4:	7
0x1FF8:	3
0x1FFC:	15
0x2000:	4
register:	2



# Implementing recursion

How can multiple instances of the same function coexist?

```

1  /*
2   recursive factorial function
3   */
4  unsigned factorial(unsigned n)
5  {
6      if (n > 0)
7          return factorial(n-1) * n;
8      else
9          return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FEC:	7
n 0x1FF0:	2
0x1FF4:	7
0x1FF8:	3
0x1FFC:	15
0x2000:	4

register:	2
-----------	---

# Implementing recursion

How can multiple instances of the same function coexist?

```

1  /*
2   recursive factorial function
3  */
4  unsigned factorial(unsigned n)
5  {
6   if (n > 0)
7   return factorial(n-1) * n;
8   else
9   return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FF4:	7
n 0x1FF8:	3
0x1FFC:	15
0x2000:	4
register:	2

# Implementing recursion

How can multiple instances of the same function coexist?

```

1  /*
2   recursive factorial function
3  */
4  unsigned factorial(unsigned n)
5  {
6   if (n > 0)
7   return factorial(n-1) * n;
8   else
9   return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FF4:	7
n 0x1FF8:	3
0x1FFC:	15
0x2000:	4
register:	6

# Implementing recursion

How can multiple instances of the same function coexist?

```

1  /*
2   recursive factorial function
3  */
4  unsigned factorial(unsigned n)
5  {
6      if (n > 0)
7          return factorial(n-1) * n;
8      else
9          return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FF4:	7
n 0x1FF8:	3
0x1FFC:	15
0x2000:	4
register:	6

# Implementing recursion

How can multiple instances of the same function coexist?

```
1  /*
2   recursive factorial function
3  */
4  unsigned factorial(unsigned n)
5  {
6      if (n > 0)
7          return factorial(n-1) * n;
8      else
9          return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FFC:	15
n 0x2000:	4

register:	6
-----------	---

# Implementing recursion

How can multiple instances of the same function coexist?

```
1  /*
2   recursive factorial function
3  */
4  unsigned factorial(unsigned n)
5  {
6      if (n > 0)
7          return factorial(n-1) * n;
8      else
9          return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FFC:	15
n 0x2000:	4

register:	24
-----------	----

# Implementing recursion

How can multiple instances of the same function coexist?

```
1  /*
2   recursive factorial function
3  */
4  unsigned factorial(unsigned n)
5  {
6      if (n > 0)
7          return factorial(n-1) * n;
8      else
9          return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

0x1FFC:	15
n 0x2000:	4
register:	24

# Implementing recursion

How can multiple instances of the same function coexist?

```
1  /*
2   recursive factorial function
3   */
4  unsigned factorial(unsigned n)
5  {
6     if (n > 0)
7         return factorial(n-1) * n;
8     else
9         return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

register: 24



# Implementing recursion

How can multiple instances of the same function coexist?

```
1  /*
2   recursive factorial function
3  */
4  unsigned factorial(unsigned n)
5  {
6      if (n > 0)
7          return factorial(n-1) * n;
8      else
9          return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

register: 24

# Implementing recursion

- The mechanism of the function calls in C is capable of writing recursive functions

# Implementing recursion

- The mechanism of the function calls in C is capable of writing recursive functions
- All the data (local variables, return addresses) of the calling functions are stored in the stack

# Implementing recursion

- The mechanism of the function calls in C is capable of writing recursive functions
- All the data (local variables, return addresses) of the calling functions are stored in the stack
- Whether the function calls itself or an other function makes no difference

# Implementing recursion

- The mechanism of the function calls in C is capable of writing recursive functions
- All the data (local variables, return addresses) of the calling functions are stored in the stack
- Whether the function calls itself or an other function makes no difference
- The maximal depth of recursive calls: given by the stack size

# Recursion or iteration – factorial

Calculating  $n!$  recursively – elegant, but inefficient

```
1 unsigned fact_rec(unsigned n)
2 {
3     if (n == 0)
4         return 1;
5     return fact_rec(n-1) * n;
6 }
```

[link](#)

# Recursion or iteration – factorial

Calculating  $n!$  recursively – elegant, but inefficient

```
1 unsigned fact_rec(unsigned n)
2 {
3     if (n == 0)
4         return 1;
5     return fact_rec(n-1) * n;
6 }
```

[link](#)

and iteratively – boring, but efficient

```
1 unsigned fact_iter(unsigned n)
2 {
3     unsigned f = 1, i;
4     for (i = 2; i <= n; ++i)
5         f *= i;
6     return f;
7 }
```

[link](#)

# Recursion or iteration – Fibonacci

Calculating  $F_n$  recursively – elegant, but way too slow!

```
1 unsigned fib_rec(unsigned n)
2 {
3     if (n <= 1)
4         return n;
5     return fib_rec(n-1) + fib_rec(n-2);
6 }
```

[link](#)



# Recursion or iteration – Fibonacci

Calculating  $F_n$  recursively – elegant, but way too slow!

```
1 unsigned fib_rec(unsigned n)
2 {
3     if (n <= 1)
4         return n;
5     return fib_rec(n-1) + fib_rec(n-2);
6 }
```

[link](#)

and iteratively – boring, but efficient

```
1 unsigned fib_iter(unsigned n)
2 {
3     unsigned f1 = 0, f2 = 1, f3, i;
4     for (i = 2; i <= n; ++i) {
5         f3 = f1 + f2;
6         f1 = f2;
7         f2 = f3;
8     }
9     return f2;
10 }
```

[link](#)

# Recursion or iteration

- 1 Every recursive algorithm can be transformed to an iterative one (loops)

# Recursion or iteration

- 1 Every recursive algorithm can be transformed to an iterative one (loops)
  - There is no general method for this transformation

# Recursion or iteration

- 1 Every recursive algorithm can be transformed to an iterative one (loops)
  - There is no general method for this transformation
- 2 Every iterative algorithm can be transformed to a recursive one

# Recursion or iteration

- 1 Every recursive algorithm can be transformed to an iterative one (loops)
  - There is no general method for this transformation
- 2 Every iterative algorithm can be transformed to a recursive one
  - Easy to do systematically, but usually not efficient

# Recursion or iteration

- 1 Every recursive algorithm can be transformed to an iterative one (loops)
  - There is no general method for this transformation
- 2 Every iterative algorithm can be transformed to a recursive one
  - Easy to do systematically, but usually not efficient

There is no universal truth: the choice between recursive and iterative algorithms depends on the problem

# Iterative algorithms recursively

## Traversing arrays recursively (without loops)

```
1 void print_array(int* array, int n)
2 {
3     if (n == 0)
4         return;
5     printf("%d ", array[0]);
6     print_array(array+1, n-1); /* recursive call */
7 }
```

# Iterative algorithms recursively

## Traversing arrays recursively (without loops)

```
1 void print_array(int* array, int n)
2 {
3     if (n == 0)
4         return;
5     printf("%d ", array[0]);
6     print_array(array+1, n-1); /* recursive call */
7 }
```

## Traversing strings recursively

```
1 void print_string(char* str)
2 {
3     if (str[0] == '\0')
4         return;
5     printf("%c", str[0]);
6     print_string(str+1); /* recursive call */
7 }
```



# Iterative algorithms recursively

## Traversing arrays recursively (without loops)

```
1 void print_array(int* array, int n)
2 {
3     if (n == 0)
4         return;
5     printf("%d ", array[0]);
6     print_array(array+1, n-1); /* recursive call */
7 }
```

## Traversing strings recursively

```
1 void print_string(char* str)
2 {
3     if (str[0] == '\0')
4         return;
5     printf("%c", str[0]);
6     print_string(str+1); /* recursive call */
7 }
```

# Printing number in a given numeral system

recursively

```
1 void print_base_rec(unsigned n, unsigned base)
2 {
3     if (n >= base)
4         print_base_rec(n/base, base);
5     printf("%d", n%base);
6 }
```

[link](#)

# Printing number in a given numeral system

recursively

```
1 void print_base_rec(unsigned n, unsigned base)
2 {
3     if (n >= base)
4         print_base_rec(n/base, base);
5     printf("%d", n%base);
6 }
```

[link](#)

iteratively

```
1 void print_base_iter(unsigned n, unsigned base)
2 {
3     unsigned d; /* power of base not greater than n */
4     for (d = 1; d*base <= n; d*=base);
5     while (d > 0)
6     {
7         printf("%d", (n/d)%base);
8         d /= base;
9     }
10 }
```

[link](#)

# When the recursive algorithm is definitely better

The array below stores a labyrinth

```

1  char lab[9][9+1] = {
2      "+-----+",
3      "|           |",
4      "+-+  ++  ++",
5      "|           |",
6      "|  +  +--+ |",
7      "|  |  |   |",
8      "+-+  +--+ |",
9      "|           |",
10     "+-----+",
11 };

```

[link](#)

# When the recursive algorithm is definitely better

The array below stores a labyrinth

```

1  char lab[9][9+1] = {
2      "+-----+",
3      "|           |",
4      "+-+  ++  ++",
5      "|           |",
6      "|  +  +--+ |",
7      "|  |  |   |",
8      "+-+  +--+ |",
9      "|           |",
10     "+-----+",
11 };

```

[link](#)

Let us visit the entire labyrinth from start position (x,y)

```

1  traverse(lab, 1, 1);

```

# When the recursive algorithm is definitely better

The array below stores a labyrinth

```

1  char lab[9][9+1] = {
2      "+-----+",
3      "|           |",
4      "+-+  +-+  +-+",
5      "|           |",
6      "|  +  +--+ |",
7      "|  |  |   |",
8      "+-+  +--+ |",
9      "|           |  |",
10     "+-----+-+"
11 };

```

[link](#)

Let us visit the entire labyrinth from start position (x,y)

```

1  traverse(lab, 1, 1);

```

We go in every possible direction and visit the yet unvisited parts of the labyrinth

# When the recursive algorithm is definitely better

The simplicity of the recursive solution is striking

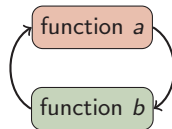
```
1 void traverse(char lab[][9+1], int x, int y)
2 {
3     lab[x][y] = '.';           /* mark that we were here */
4     if (lab[x-1][y] == ' ') /* go upwards, if needed */
5         traverse(lab, x-1, y);
6     if (lab[x+1][y] == ' ') /* go downwards, if needed */
7         traverse(lab, x+1, y);
8     if (lab[x][y-1] == ' ') /* go left, if needed */
9         traverse(lab, x, y-1);
10    if (lab[x][y+1] == ' ') /* go right, if needed */
11        traverse(lab, x, y+1);
12 }
```

[link](#)

It is also possible to do with an iterative algorithm – but it is much more complex

# Indirect recursion

Indirect recursion: Functions mutually call each other



```
1  /* forward declaration */
2  void b(int); /* name, return type, parameter types */
3
4  void a(int n) {
5      ...
6      b(n); /* b can be called due to the forward decl. */
7      ...
8  }
9
10 void b(int n) {
11     ...
12     a(n);
13     ...
14 }
```



# Forward declaration

Forward declaration will be necessary for recursive data structures

```
1  /* forward declaration */
2  struct child_s;
3
4  struct mother_s { /* mother type */
5      char name[50];
6      struct child_s *children[20]; /*pntr. arr. of children*/
7  };
8
9  struct child_s { /* child type */
10     char name[50];
11     struct mother_s *mother; /*pointer to the mother*/
12 };
```

## Chapter 2

### Multi-dimensional arrays

# Multi-dimensional arrays

**1D array** Elements of the same type, stored in the memory beside each other

# Multi-dimensional arrays

**1D array** Elements of the same type, stored in the memory beside each other

**2D array** 1D arrays of the same size and same type, stored in the memory beside each other

# Multi-dimensional arrays

- 1D array** Elements of the same type, stored in the memory beside eachother
- 2D array** 1D arrays of the same size and same type, stored in the memory beside eachother
- 3D array** 2D arrays of the same size and same type, stored in the memory beside eachother

# Multi-dimensional arrays

- 1D array Elements of the same type, stored in the memory beside eachother
- 2D array 1D arrays of the same size and same type, stored in the memory beside eachother
- 3D array 2D arrays of the same size and same type, stored in the memory beside eachother

... ..

# Two-dimensional

## ■ Declaration of a 2D array:

```
1 char a[3][2]; /* 3row x 2column array of characters */  
2             /* 3-sized array of 2-sized 1D arrays */
```

a[0][0]	a[0][1]
a[1][0]	a[1][1]
a[2][0]	a[2][1]

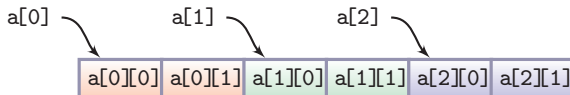
# Two-dimensional

## ■ Declaration of a 2D array:

```
1 char a[3][2]; /* 3row x 2column array of characters */  
2             /* 3-sized array of 2-sized 1D arrays */
```

a[0][0]	a[0][1]
a[1][0]	a[1][1]
a[2][0]	a[2][1]

## ■ In C language, storage is done row by row (the second index changes quicker)





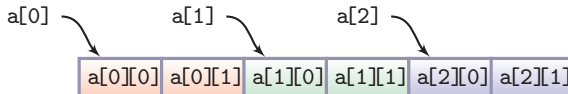
# Two-dimensional

## ■ Declaration of a 2D array:

```
1 char a[3][2]; /* 3row x 2column array of characters */  
2             /* 3-sized array of 2-sized 1D arrays */
```

a[0][0]	a[0][1]
a[1][0]	a[1][1]
a[2][0]	a[2][1]

- In C language, storage is done row by row (the second index changes quicker)



- a[0], a[1] and a[2] are 2-sized 1D arrays

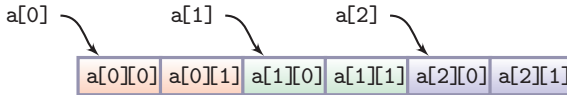
# Taking a 2D array row by row

## ■ Filling a 1D array (row) with the given element

```
1 void fill_row(char row[], size_t size, char c)
2 {
3     size_t i;
4     for (i = 0; i < size; ++i)
5         row[i] = c;
6 }
```

## ■ Filling a 2D array row by row

```
1 char a[3][2];
2 fill_row(a[0], 2, 'a'); /* row 0 is full of 'a' */
3 fill_row(a[1], 2, 'b'); /* row 1 is full of 'b' */
4 fill_row(a[2], 2, 'c'); /* row 2 is full of 'c' */
```



# Taking a 2D array as one entity

- taking as a 2D array – only if number of columns is known

```
1 void print_array(char array[][2], size_t nrows)
2 {
3     size_t row, col;
4     for (row = 0; row < nrows; ++row)
5     {
6         for (col = 0; col < 2; ++col)
7             printf("%c", array[row][col]);
8         printf("\n");
9     }
10 }
```

- Usage of the function

```
1 char a[3][2];
2 ...
3 print_array(a, 3);      /* printing a 3-row array */
```

# Taking a 2D array as one entity

## ■ taking 2D array as a pointer

```
1 void print_array(char *array, int nrows, int ncols)
2 {
3     int row, col;
4     for (row = 0; row < nrows; ++row)
5     {
6         for (col = 0; col < ncols; ++col)
7             printf("%c", array[row*ncols+col]);
8         printf("\n");
9     }
10 }
```

## ■ Usage of the function

```
1 char a[3][2];
2 ...
3 print_array((char *)a, 3, 2); /* 3 rows 2 columns */
```

# Dynamic 2D array

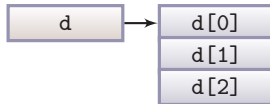
Let's allocate memory for a 2D array. We would like to use the conventional way of indexing for the array `d[i][j]`

d

```
1 double **d = (double**) malloc (3*sizeof(double*));  
2 d[0] = (double*) malloc (3*4*sizeof(double));  
3 for (i = 1; i < 3; ++i)  
4     d[i] = d[i-1] + 4;
```

# Dynamic 2D array

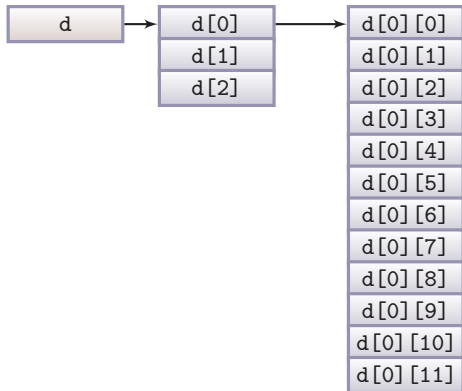
Let's allocate memory for a 2D array. We would like to use the conventional way of indexing for the array `d[i][j]`



```
1 double **d = (double**) malloc (3*sizeof(double*));
2 d[0] = (double*) malloc (3*4*sizeof(double));
3 for (i = 1; i < 3; ++i)
4     d[i] = d[i-1] + 4;
```

# Dynamic 2D array

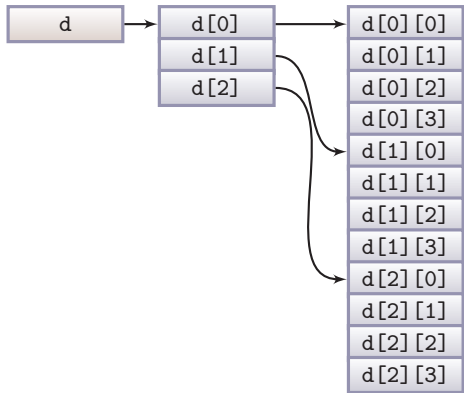
Let's allocate memory for a 2D array. We would like to use the conventional way of indexing for the array `d[i][j]`



```
1 double **d = (double**) malloc (3*sizeof(double*));
2 d[0] = (double*) malloc (3*4*sizeof(double));
3 for (i = 1; i < 3; ++i)
4     d[i] = d[i-1] + 4;
```

# Dynamic 2D array

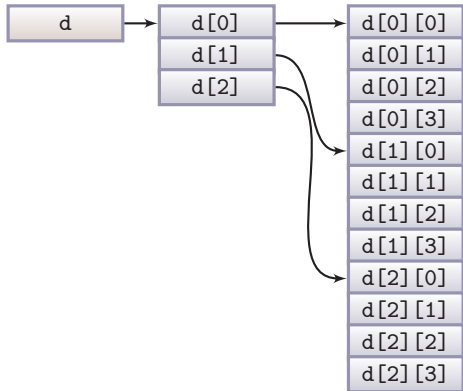
Let's allocate memory for a 2D array. We would like to use the conventional way of indexing for the array  $d[i][j]$



```
1 double **d = (double**) malloc (3 * sizeof (double*));
2 d[0] = (double*) malloc (3 * 4 * sizeof (double));
3 for (i = 1; i < 3; ++i)
4     d[i] = d[i-1] + 4;
```



# Dynamic 2D array



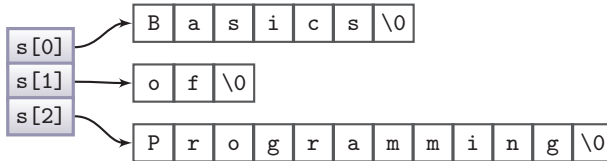
Releasing the array

```
1 free(d[0]);  
2 free(d);
```

# Array of pointers

## ■ Defining an array of pointers and passing it to a function

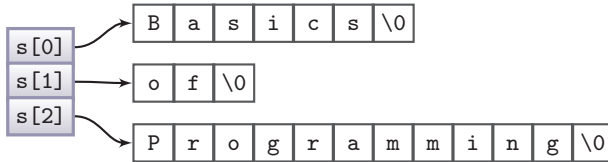
```
1 char *s[3] = {"Basics", "of", "programming"};  
2 print_strings(s, 3);
```



# Array of pointers

## ■ Defining an array of pointers and passing it to a function

```
1 char *s[3] = {"Basics", "of", "programming"};  
2 print_strings(s, 3);
```



## ■ Taking an array of pointers with a function

```
1 void print_strings(char *strings[], size_t size)  
2 /*           char **strings is also possible */  
3 {  
4     size_t i;  
5     for (i = 0; i < size; ++i)  
6         printf("%s\n", strings[i]);  
7 }
```

Thank you for your attention.