Multidimensional Arrays – Recursion – Union

Basics of Programming 1

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

29 October, 2025

Content

Chapter 1

Multi-dimensional arrays

Multi-dimensional arrays

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

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

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- Declaration of a 2D array:

```c
char a[3][2]; /* 3row x 2column array of characters */
              /* 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]		a[2]	
a[0][0]	a[0][1]	a[1][0]	a[1][1]	a[2][0]	a[2][1]

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

Taking a 2D array row by row

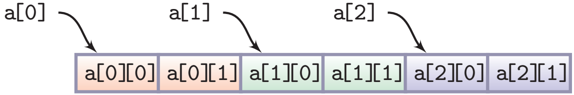
DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

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

```
void fill_row(char row[], size_t size, char c)
{
  size_t i;
  for (i = 0; i < size; ++i)
    row[i] = c;
}
```

- Filling a 2D array row by row

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



Taking a 2D array as one entity

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

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

```c
void print_array(char array[][2], size_t nrows)
{
  size_t row, col;
  for (row = 0; row < nrows; ++row)
  {
    for (col = 0; col < 2; ++col)
      printf("%c", array[row][col]);
    printf("\n");
  }
}
```

- Usage of the function

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

Taking a 2D array as one entity

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- taking 2D array as a pointer

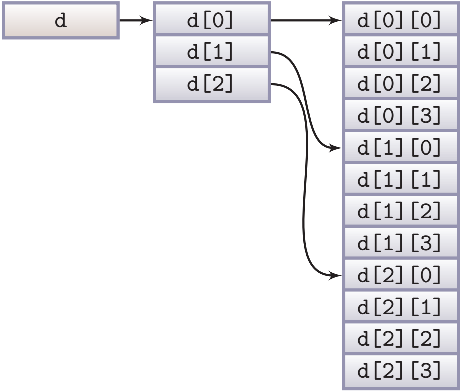
```c
void print_array(char *array, int nrows, int ncols)
{
  int row, col;
  for (row = 0; row < nrows; ++row)
  {
    for (col = 0; col < ncols; ++col)
      printf("%c", array[row*ncols+col]);
    printf("\n");
  }
}
```

- Usage of the function

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

Dynamic 2D array

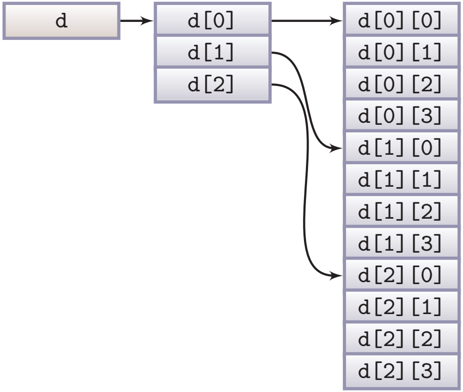
Let's allocate memory for a 2D array. We would like to use the conventional way of indexing for the array `d[i][j]`




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

Dynamic 2D array

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES



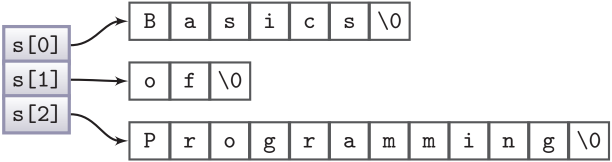
Releasing the array

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

Array of pointers

- Defining an array of pointers and passing it to a function

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



- Taking an array of pointers with a function

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

Chapter 2

Recursion and recursive algorithms

Recursion – definition

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

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

- 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

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

Recursion –outlook

Several everyday problems can be formulated recursively

- 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

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

Let us implement it to C!

```c
unsigned factorial(unsigned n)
{
  if (n > 0)
    return factorial(n-1) * n;
  else
    return 1;
}
```

- Calling the function

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

Some considerations

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- How to imagine recursive functions?

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

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

Implementing recursion

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

How can multiple instances of the same function coexist?

```c
/*
  recursive factorial function
 */
unsigned factorial(unsigned n)
{
  if (n > 0)
    return factorial(n-1) * n;
  else
    return 1;
}

int main(void)
{
  ...
   factorial(4);
  ...
}
```

register: 24

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

```c
unsigned fact_rec(unsigned n)
{
  if (n == 0)
    return 1;
  return fact_rec(n-1) * n;
}
```

link

and iteratively – boring, but efficient

```c
unsigned fact_iter(unsigned n)
{
  unsigned f = 1, i;
  for (i = 2; i <= n; ++i)
    f *= i;
  return f;
}
```

link

Recursion or iteration – Fibonacci

Calculating F_n recursively – elegant, but way too slow!

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

link

and iteratively – boring, but efficient

```
unsigned fib_iter(unsigned n)
{
  unsigned f1 = 0, f2 = 1, f3, i;
  for (i = 2; i <= n; ++i) {
      f3 = f1 + f2;
      f1 = f2;
      f2 = f3;
  }
  return f2;
}
```

link

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)

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

Traversing strings recursively

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

Printing number in a given numeral system

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

recursively

```c
void print_base_rec(unsigned n, unsigned base)
{
  if (n >= base)
    print_base_rec(n/base, base);
  printf("%d", n%base);
}
```

link

iteratively

```c
void print_base_iter(unsigned n, unsigned base)
{
  unsigned d; /* power of base not greater than n */
  for (d = 1; d*base <= n; d*=base);
  while (d > 0)
  {
    printf("%d", (n/d)%base);
    d /= base;
  }
}
```

link

When the recursive algorithm is definitely better

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

The array below stores a labyrinth

```
char lab[9][9+1] = {
  "+-------+",
  "|       |",
  "+-+ ++ ++",
  "|       |",
  "| + +-+ |",
  "| | |   |",
  "+-+ +-+ |",
  "|     | |",
  "+-----+-+"
};
```

link

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

```
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

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

The simplicity of the recursive solution is striking

```
void traverse(char lab[][9+1], int x, int y)
{
  lab[x][y] = '.';          /* mark that we were here */
  if (lab[x-1][y] == ' ') /* go upwards, if needed */
    traverse(lab, x-1, y);
  if (lab[x+1][y] == ' ') /* go downwards, if needed */
    traverse(lab, x+1, y);
  if (lab[x][y-1] == ' ') /* go left, if needed */
    traverse(lab, x, y-1);
  if (lab[x][y+1] == ' ') /* go right, if needed */
    traverse(lab, x, y+1);
}
```

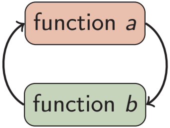
link

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

Indirect recursion

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

Indirect recursion: Functions mutually call each other




```c
/* forward declaration */
void b(int); /* name, return type, parameter types */

void a(int n) {
  ...
  b(n); /* b can be called due to the forward decl. */
  ...
}

void b(int n) {
  ...
  a(n);
  ...
}
```

Forward declaration

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

Forward declaration will be necessary for recursive data structures

```c
/* forward declaration */
struct child_s;

struct mother_s { /* mother type */
  char name[50];
  struct child_s *children[20]; /*pntr. arr. of children*/
};

struct child_s { /* child type */
  char name[50];
  struct mother_s *mother; /*pointer to the mother*/
};
```

Chapter 3

Union and bitfield

Union data type

Union

Simple data type capable of storing data of different types

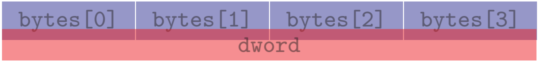
```c
union data {
  short int i;  /* overlapped memory layout !!! */
  double d;
  char str[20];
};
```


```c
union data a;
strcpy(a.str, "Hello world");
printf("%f", a.d);  /* first 8 bytes as a double */
```

The size of the type is determined by the longest member

Typical application


```c
union data {
  unsigned char bytes[4];
  unsigned int dword;
};
```




```c
union data a;
a.dword = 234568;
printf("%u", a.bytes[2]);
```

The sample code is correct only if the size of `unsigned int` is at least 32 bits

An other typical application

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```c
typedef struct { double x1, x2, y1, y2; } line_t;
typedef struct { double x0, y0, r; } circle_t;

typedef struct {
  enum {LINE, CIRCLE} type; /* what is inside */
  union { /* this part is EITHER a line OR a circle */
    line_t line;
    circle_t circle;
  };
} object_t;
```

```c
object_t array[4];
```


```c
array[0].type = LINE;
array[0].line.x1 = 2;
```

Example

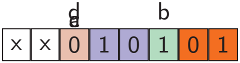
L	line	C	circle	L	line	L	line

```c
for (i = 0; i < 4; ++i) {
  if (array[i].type == LINE) {
    line_t line = array[i].line;
    /* process line */
  }
  else if (array[i].type == CIRCLE) {
    circle_t circle = array[i].circle;
    /* circle processing */
  }
}
```

Bitfield data type

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

In low level programming it is sometimes useful to work with the bits of a data as individual variables.



Bitfield

In a single variable we store several variables.

```c
struct status {
  unsigned a : 2;
  unsigned b : 1;
  unsigned c : 2;
  unsigned d : 1;
};
```

```c
struct status st1;
st1.a = 1;
st1.b = 1;
st1.c = 2;
st1.d = 0;
```

Bitfields can have only `unsigned int` or `int` members

Thank you for your attention.