Low-level programming

Lecture 6

Functions

Marcin Radom, Ph.D.

Institute of Computing Science
Faculty of Computing and Telecommunication

Functions

- In general, functions allow the division of a code into smaller subunits.
- They contain some smaller fragments of a solution, which possibly can be used for different problems (e.g., function giving a maximum value from an array).
- Program divided into smaller modules can be debugged easier.
- Functions in C are divided into the ones returning some value as a result of their actions and the ones which don't directly return anything.
- Forbidden operations in C:
  - Calling for an undeclared function.
  - Creation of a nested function (i.e., a function is declared inside some other function).
- **Function declaration:**

```
type   function_identifier
          ( function_arguments);
```

  - *type*: **numerical, pointer, reference, structure**
  - *function_argument:* `type identifier`

Functions

- **Function signature** is a name of a function, a list of its arguments and their types.
- **Example:**

```
// declaration and definition of a function must be compatible!!!
int f1( int x, int y );
float value( float price, int number);
char* answer( char *question);

// error: each argument must have its type, we cannot however use
// comma-declaration for function arguments:
double alfa( double x1, x2 );

max( int k, int l );      // default type int

void print( char *string);  // no value returned (void)
```

Functions

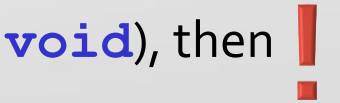
- **Function definition:**

```
type  function_identifier ( list_of_arguments )
{  instructions / body of a function }
```

  - *function block*: a sequence of instructions (any):
    - to the end of block
    - until instruction `return`
  - *formal argument* : `type identifier`
- Formal arguments are local variables, they are „private property" of a function (similarly as variables declared inside the function block).
- The simplest and smallest (yet with proper syntax) function:

```
example () { }  // takes nothing, gives nothing, safest function possible!
```

- If we do not provide returned type (neither we use keyword `void`), then in C it is assumed that such a function returns type `int`.



Functions – *return*

- Instruction **`return`** returns something from a function – any statement which can be calculated is valid:

```
return statement;
```

- Statement will be transformed into some value of a given type (type in function definition).
- Instruction:

```
return;
```

stops the function but returns no value.

- There can be more than one **`return`** instruction inside a function.
- There is no formal/syntax error when in one place of a function there is **`return`** with something, and in the other place **`return`** without any value – however it can cause trouble!
- If a function returns no value (but it is not void) then its returned values are random ones.

Functions

- **Example:**

```
float sqr( float x )
{
     return x * x ;
}

void max( int *m, int a, int b )
{
     *m = a > b ? a : b ;
}
// function end with its last instruction before last bracket: }
// same effect as: return; - possible only when void

void PrintResult ( float x )
{
     printf ( "\nResult : %10.2f" , x);
}
```

Functions

- **Example:**

```
// divides text in array S in such a way that characters in S with even
// indexes are being put into array D2, the one with odd indexes – into D1
int split( char *S, char *D1, char *D2 )
{
     // counts characters in S until sign '\0'
     int count_S = 0;   // local variable
     while ( *S )
     {              // odd index characters in S
          if ( count_S & 0x01 )
               *D2++ = *S++;
          else
               *D1++ = *S++;
           count_S ++;
     }
     // adds special character '\0' to D1 and D2
     *D1 = *D2 = 0;
     return count_S;
}
```

Calling function

- **Syntax:**

  function_identifier(list_of_arguments)

  - Where **actual argument** (when calling a function) can also be a statement.

- **Example:**

```
a = 1 + sqr( 1.25 );      // calling in a statement
max( &better, price_1, price_2);//calling as a separate instruction
sqr(3 + b * d );              // actual argument is a statement
PrintResult ((sqrt(delta) - 4 * a * c ) / ( 2 * a ) ) ;
printf( "\n%d\t%d\n", a + 2, a - 1 );
```

- Formal arguments of a function obtain their values from actual arguments used when calling a function.
- Formal arguments are **copies** of data from calling instruction, their modifications inside a function does not change the original variables used in calling order.

Functions – actual and formal arguments

- **Example:**

```
int Funa ( int a, int b, int c )
{
      if ( a > 3 * b – 1)
          return c + 2;
      else
          return c * c + 1;
}
// .........
int k, r;
// .........
int x = Funa (12, k + 4, r - 3); // calling function
 // a = 12,  b = k + 4,  c = r - 3
```

Functions

- If a function declaration and definition are not the same, and the function itself is defined in another file (other than its declaration) then the compiler may not notice such a mistake. If a function is defined in the same file as its declaration, the compiler will see this as an error.
- It means that, e.g., function from another file than the one where it is called returns type **double** and a programmer tries to write the result into type **int** - the compiler will not protest while compiling the program.
- As a result the program will start (from this calling/writing results moment) to use some random results – such an error it often difficult to detect.
- If in declaration types are not given (nor the arguments), e.g.:

```
double atof();
```

then nothing is assumed about the arguments – all validity control for the argument cannot work.

Functions - Example

```
/* atof: characters array converted into double type variable */
double atof(char s[]){
     double val, power;
     int i, sign;
     for (i = 0; isspace(s[i]); i++)
          ; /* omitt white signs */
     sign = (s[i] == '-') ? -1 : 1;
     if (s[i] == '+' || s[i] == '-’)
          i++;
     for (val = 0.0; isdigit(s[i]); i++)
          val = 10.0 * val + (s[i] – '0’);
     if (s[i] == '.’)
          i++;
     for (power = 1.0; isdigit(s[i]); i++) {
          val = 10.0 * val + (s[i] – '0’);
          power *= 10;
     }
     return sign * val / power;
}
// for example for 5 signs: 12.34 value val
// will be 1234.0,
// and for variable power 100.0.
// The results is 1234.0 / 100.0 = 12.34 */
```

```
12.34
Czesc liczby przed przecinkiem:
s[0] = 1, val = 1
s[1] = 2, val = 12
Czesc liczby po przecinku:
s[3] = 3, val = 123, power = 10
s[4] = 4, val = 1234, power = 100
Wynik = 12.34
```

Functions - Example

```
/* atoi: conversion to integer using function atof */
int atoi(char s[])
{
     double atof(char s[]);// prototype of atof
     return (int) atof(s);  // casting result into int
}
```

- In such a function **atof()** is used to change array into `double` value (as in previous page of this presentation), then the result is converted into `int`.
- There may be losses during such a conversion (e.g. fraction).
- There is however an explicit casting: **(`int`)** so the compiler assumes that a programmer knows what he is doing.

- **Giving argument to a function:**
  - Formal and actual arguments comply when:
    - Types are identical
    - Conversion of types is possible

Sending arguments to the function

Problems and examples

Sending argument into a function

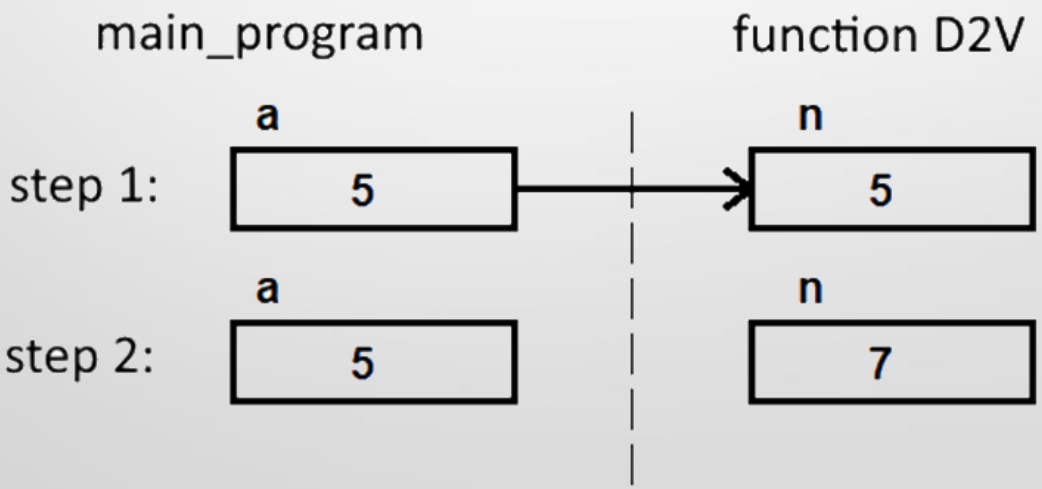
- **Example:**

```
long sum( long x )
{
      return x + 1000;
}

sum(3);          // correct int -> long
sum(1543443L);   // correct without conversion
sum(17.8);       // risky one: double -> long, accuracy loss possible
sum('A');        // correct char -> long
sum("ALA");      // error, conversion impossible

int count( float *wf );

int *wi;
count( wi );   // error, wrong pointer
count(( float*) wi);  // correct, yet risky one
```

Sending argument into a function

- There are many ways of sending the arguments:
  - **giving by value**

```
void D2V ( int n ) // sending value only
{
      n = n + 2;    // step 2
}
int a = 5;
D2V ( a );          // step 1
```



Sending argument into a function

- In C language arguments are (by default) given as values (meaning: as a copy).
- Because of that, it is not possible for a function to change the original values directly.
- E.g., if we had a function changing places for two values (function **swap**) and we called it like this:

```
swap(a, b);
```

And if the function *swap* has been declared as follows:

```
void swap(int x, int y) { // ...
     int tmp = x;
     x = y;
     y = tmp;
}
```

- The the function *swap* has no access to the original values of **a** and **b** (because they are given to it as values/copies) so it can only change its own private variables: **x** and **y**.

Sending argument into a function

- In other words: whatever a function does with its arguments, the original variables providing values to the arguments are safe (and stored somewhere else).
- The only way to change the original values is to use pointers, then calling of a function is realized in such a way:

```
swap(&a, &b);
```

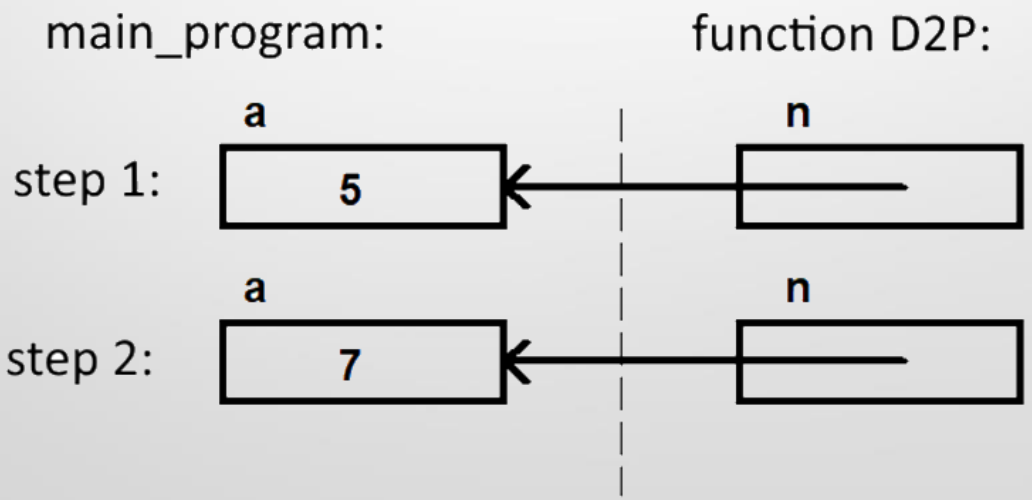
- Operator **&** provides address of a variable, so **&a** is an address of *a*.
- Correct function definition:

```
void swap(int *x, int *y)
{   // changes places of values *x and *y
    int tmp = *x;
    *x = *y;
    *y = tmp;
}
```

Sending argument to a function

- **sending a pointer**

```
void D2P( int *n )    // providing pointer
{
    *n = *n + 2;                  // step 2
}
int a = 5;
D2P( &a );                       // step 1
```



Pointer to the constant variable vs. constant pointers

- **Pointer to the constant variable** is a pointer which „**thinks**" that its pointed variable is a constant, so the pointer will not allow any changes.

```
const long *ws_st;              // pointer to a constant
const long distance = 5786;
ws_st = & distance;             // set pointer to the variable
long how_far = *ws_st;          // correct, how_far == distance

// error, pointer cannot modify its variable:
*ws_st = 1298;

long war = 10, something;
ws_st = & war;
war = 150;                      // correct
something = *ws_st;             // correct, something= 150

// error, constant pointer cannot be used to modify its variable:
*ws_st = 150;
```

Pointer to the constant variable vs. constant pointers

- **Constant pointer**
  - It is a pointer which points to the same variable for the whole time. Its address cannot be changed.

```
float price = 12.5, net_value;
float *const swx = & price ;
   // constant pointer, initial value (address) is REQUIRED
price = 15.8;                // correct
*swx = 15.8;                 // correct

// error, this pointer cannot change its address:
swx = &net_value;
```

Sending argument into a function

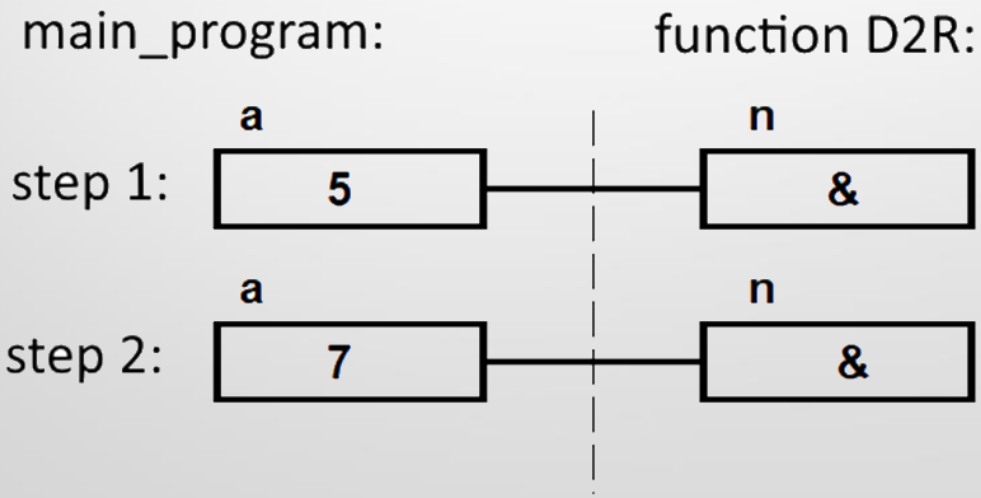
- **sending pointer (cont.)**
  - For constant argument:

```
// lpi is a pointer to constant variable
float CircleField(const float *lpi, float *pr)
{
      // error, this pointer cannot be used to change the value
      // of its variable
      *lpi = 2.5;
      /* ... */
}
```

Sending argument into a function

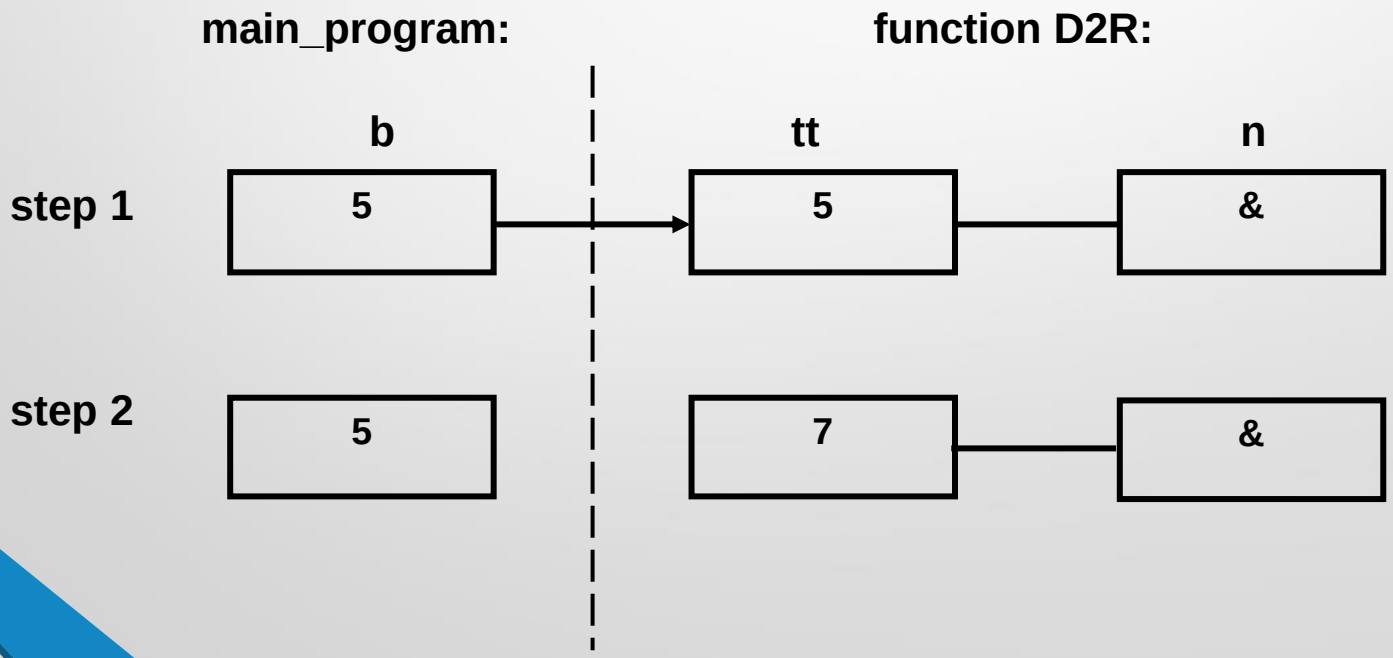
- **sending reference**

```
void D2R(int& n)    // sending a reference
{
    n = n + 2;      // step 2
}
int a = 5;
D2R( a );           // step 1
/* address of variable a is being sent. It is useful for bigger objects
however may cause problems if done incorrectly */
```



Sending argument into a function

```
long b = 5;
D2R( b );                          // step 1
/* compile error, cannot convert from long to int&.

error: invalid initialization of reference of type 'int&' from expression
of type 'long int'

If that parameter was sent by value, it would be ok. */
```



Sending a structure into a function

- **Structure can be send to function as follows:**
  - **Sending fields separately:**
    - E.g., a function takes two coordinates and creates structure *point* from them:

```
struct point makepoint(int x, int y)
{
     struct point tmp;
     tmp.x = x;
     tmp.y = y;
     return tmp;
}
```

    - As we can see, there is no conflict in names because x – is a local variable and tmp.x (a field of tmp) are different variables.

Sending a structure into a function

- **Example:**

```
struct rect screen;        // reserve space for rect
struct point middle;       // reserve space for point
struct point makepoint(int, int);    // function prototype
screen.pt1 = makepoint(0,0); // initiating structure by calling function
screen.pt2 = makepoint(XMAX, YMAX);  // same as above

middle = makepoint((screen.pt1.x + screen.pt2.x)/2, (screen.pt1.y +
screen.pt2.y)/2);
// coordinates of middle point of a rectangle
```

- **Sending whole structure:**
  - Structures are send as normal variables – <u>by values</u>:

```
struct point addpoint(struct point p1, struct point p2){
      p1.x += p2.x;  // p1 is a copy of some structure inside addpoint
      p1.y += p2.y;  // same as above!
      return p1;     // returns result
}
```

Sending a structure into a function

- In the next example function *Store(…)* does not have access to object AB of structure Account (because it is send by value), so it will only modify its local copy.

```
struct Account{
    long number;
    long sum;
};
void Store(struct Account kk, long amount){
     kk.sum += amount;
}
struct Account AB = { 84404, 0 };
Store( AB, 3285 );   //  AB.sum  == 0, no change
```

- This function as a result returns a structure with modified fields:

```
struct Account Store (struct Account kk, long amount){
    kk.sum += amount;
    return kk;
}
AB = Store ( AB, 3285 );  //  AB.sum == 3285
```

Sending a structure into a function

- **Sending pointer of a structure:**
  - If we want to send some large structure into the function, then using a pointer is usually better than creating a (large) copy in memory of this structure.
  - Pointers pointing to structure are by all means normal pointer variables.
  - **Example:**

```
struct information {
        int x, y, z;
};

void subtract (struct information *info, int px, int py, int pz)
{
        info->x -= px;
        info->y -= py;
        info->z -= pz;
}
```

Functions – Example 1

```
/* Program calculating tax:
I   degree: up to 6000 PLN -> tax 0 PLN
                           above 6000 -> tax 10% of amount
II degree: up to 5000 PLN -> tax 25 PLN
                        above 5000 PLN -> tax 25 PLN + 15% from surplus
III degree : tax is 20% of total amount
*/

     int St1 (int Amount){ // I degree
          if (Amount <= 6000)
               return 0;
          else
               return int(Amount * 0.1); // 10% of amount
     }
     int St2 (int Amount){ // II degree
          if (Amount <= 5000)
               return 25;
          else
               return int(25 + (Amount - 5000) * 0.15);
     }
     int St3(int Amount ){ // III degree
         return int(Amount * 0.2); // 20% of total amount
     }
```

Functions – Example 1

```
int main() {
     int Donation, Tax, Degree= -1;
     printf("Provide amount of a donation: ");
     scanf("%d", &Donation);
     while (Degree < 1 || Degree> 3){
          printf("\nProvide relative degree [1, 2, 3] : ");
          scanf("%d", &Degree);
     }
     switch (Degree){
          case 1 : Tax = St1(Donation); break;
          case 2 : Tax = St2(Donation); break;
          case 3 : Tax = St3(Donation); break;
     }
     printf("\nTax is equal to %d PLN.\n\n", Tax);
     return 0;
}
```

Functions – Example 2

A program calculating the formula:

$$R = \sum_{i=1}^{n} \prod_{j=1}^{m} \frac{F1(i,j) - F2(i,j)}{F1^2(j+2, i+3) + 1}$$

where:

$$F1(i,j) = \frac{\sin^2 i + \cos^2 j}{i+j} \quad \text{when} \quad i \geq j$$

$$F1(i,j) = \frac{\cos^3 i + \sin^3 j}{2i+j} \quad \text{when} \quad i < j$$

$$F2(i,j) = \frac{3i + 2j + 1}{2i + 5i + 4}$$

Functions – Example 2

```
/* Sum of products */
double F1(int i, int j ){
     if (i >= j)
          return (pow(sin((double)i), 2) + pow(cos((double)j), 2)) / (i + j);
     else
          return (pow(cos((double)i), 3) + pow(sin((double)j), 3)) / (2 * i + j);
}

double F2(int i, int j){
     return (3 * i + 2 * j + 1) / (2 * i + 5 * j + 4.0);
}

double Product(int i, int m){
     double ilo = 1.0;
     for (int j = 1; j <= m; ++j)
          ilo *= (F1(i,j) - F2(i,j))/ (pow(F1(j + 2, i + 3), 2) + 1);
     return ilo;
}
```

Functions – Example 2

```
int main(){
     int n, m;
     double R = 0;

     printf("Enter values for n and m : ");
     scanf("%d%d", &n, &m);

     // sum of products from i = 1 to n
     for (int i = 1; i <= n; ++i)
         R += Product(i, m);

     printf("R = %.3lf", R);
     printf("\n\n");
     return 0;
}
```

Arrays as function arguments

- If the argument of a function is an array, then a function get the ADDRESS to the first element of the array (an in this way to the whole array, because array name is a hidden pointer in C). Therefore function works on the original array (not its local copy).
- Inside a function an array usually gets another name (name of the function argument) however it is still the same array from the same area of memory.
- Array name is a hidden pointer to the first array element (to its address).
- **Example:** provide text length:

```
#include <stdio.h>
int length(char *);//prototype
int main(){
      int n = length("Ahoj!");
      printf("%d\n", n);
      return 0;
}
```

Arrays as function arguments

```
int length(char *s){
     int n;
     for(n=0; *s != '\0'; s++)
          n++;
     return n;
}
```

- Increasing variable **s** is valid – it is a pointer variable. Operation *s++* does not change the original address, it works on the private copy of the address.
- Example:

```
length(char ("Hello world!"); // constant string
length(char (tab); // char tab[100];
length(char (wsk); // char *wsk;
```

- For the function we can send only a part of the array, giving a pointer / address to the first sub-array element we choose, e.g. ***f(&a[3])*** will sent subarray of **a** starting from element [3] to the last one.

Arrays as function arguments

- Into the function we can send only a part of an array, giving a pointer / address to the first sub-array element we choose, e.g. ***f(&a[3])*** will sent subarray of **a** starting from element [3] to the last one
- On some compilers the following code will work, mostly because we have to remember that the array is a solid block of cells in the memory, so even when we do not see all of them, they are there.
  - Nevertheless, it is a „don't do it", absurd example:

```
int fun(int tab[]) {
   printf("Value: %d\n", tab[-1]); // !!!
}
...
int arr[] = {0,1,2,3,4,5}
fun(&arr[2]); // shows: „1"
```

Arrays as function arguments and sizeof

- Size of the array can be successfully established using sizeof **before we send the array into the function!**
- When it happens, inside a function the array is „reduced" to the pointer, which cannot be used with sizeof.

```
int fun(int tab[]) {
   printf("Size: %d\n", sizeof(tab) / sizeof(tab[0]) ); // !!!
}
...
int arr[] = {0,1,2,3,4,5}
fun(arr); // shows... some strange numbers OR:
//  sizeof on array function parameter will return size of 'int *'
// instead of 'int []' [-Wsizeof-array-argument]
```

- A solution to this problem is to learn the size of the array **before** in a context where an array name is still... an array ☺ and then we pass this value to the function as yet another argument.

Arrays as function arguments

- **One dimensional arrays declaration:**

```
// Declarations below are equivalent
// information to the compiler that we give a pointer is preferred
int FT(int *);          // declarations

// we inform the compiler that we send an array to the function.
// Address of its first element will be provided by array name
int FT(int[ ]);

// same effect as above, information about array size is additional and not
needed
int FT(int[10]);

int FT(int *T) { ... }  // definitions
int FT(int T[ ]) { ... }
int FT(int T[10]) { ... }
//  T [ 3 ]
//  T  +   3   *   sizeof (  int  )
```

Arrays as function arguments

- **One dimensional array, example:**

```
void ZAP1( int T[100] ){
   for ( int i = 0; i < 100; ++i )
       T[ i ] = i;
}

int ALFA[15];
ZAP1 ( ALFA ) ;        // array range exceeded
int BETA[100];
ZAP1 ( &BETA [0] );    // write whole array
int GAMMA[150];
ZAP1 ( GAMMA );        // write 2/3 if array
```

- Proper definition of the above function:

```
void ZAP2(int T[ ], int size){ // we provide the size of the array and
         // the address of its first element
   for ( int i = 0; i < size; ++i )
       T[i] = i;
}
```

Arrays as function arguments Example 1

```
void F1(int T[50]){
    for(int i = 0; i < 50; ++i)
         T[i] = i;
}

int main(){
     int x;
     int T1[70] = {0};
     int T2[10];
     F1(T2);

     x = 0;
     return 0;
}
// program will compile, there will be however execution error
// (program will end unexpectedly)
```

Arrays as function arguments

- **Multi dimensional arrays:**
  - If we provide such an array to function: `int daytab[2][13]` then we have to provide the number of columns.
  - Number of rows is not required, because an address to the first element will be provided automatically.
  - Declaration of a function accepting multi-dimensional arrays can have forms:
    - `fun(int daytab[2][13]) { … }`
    - `fun(int daytab[][13]) { … }` – number of rows is not required

Arrays as function arguments

- **Multi dimensional arrays, Example:**

```
void ZAP3( float TT[ ][10], int rows )
{
   for (int i = 0; i < rows; ++i)
       for (int j = 0; j < 10; ++j)
            TT[i][j] = i + j;
}

//  calling to element of array TT [ 3 ] [ 5 ]
//  TT + 3 * 10 * sizeof ( float ) + 5  * sizeof ( float )
```

Arrays as function arguments Example 2

- **Program with list of TVs**

```
const int MAX = 100;
struct TV{
     char Mark[32];
     int Price;
     int Number;
};
// new TV, their number must be less than  MAX

void Op_N(struct TVTab[ ], int &where){
     if (where < MAX){
          printf("Provide name: ");
          scanf("%31s", Tab[where].Mark);
          printf("Provide price: ");
          scanf("%d", &Tab[where].Price);
          printf("Provide number of TVs: ");
          scanf("%d", &Tab[where++].Number);
    } else
          printf("Table full.\n");
}
```

Arrays as function arguments Example 2

```
// show informations about TVs in store
void Op_W(struct TV Tab[ ], int end) {
     for(int i = 0; i < end ; ++i)
          printf("%d. TV: %s, Price: %d, Number in stock: %d\n", i, Tab[i].Mark,
Tab[i].Price, Tab[i].Number);
}

// Remove TV by its index
void Op_U(struct TV Tab[ ], int &end) {
     int which;
     printf("Provide TV ID: ");
     scanf("%d", &which);
     if (which >= 0 && which < end) {
          Tab[which] = Tab[--end];
          printf("Removed.\n");
     } else printf("Wrong ID.\n");
}
```

Arrays as function arguments Example 2

```
// compute sum of values of TVs in stock
void Op_S(struct TV Tab[ ], int end) {
     int Sum = 0;
     for (int i = 0; i < end ; ++i)
          Sum += Tab[i].Price * Tab[i].Number;
     printf("Total value: %d\n", Sum);
}

int main() {
     struct TV TabTel[MAX];
     int how_many = 0;
     bool go_on = true;
     char option;

     while(go_on) {
          printf("Choose option [N, W, U, S, Q] : ");
          fflush(stdin);
          scanf("%c", &option);
```

Arrays as function arguments Example 2

```
        switch(option  & 0x5F){
            case 'N':
                Op_N(TabTel, how_many);
                break;
            case 'W':
                Op_W(TabTel, how_many);
                break;
            case 'U':
                Op_U(TabTel, how_many);
                break;
            case 'S':
                Op_S(TabTel, how_many);
                break;
            case 'Q':
                go_on = false;
                break;
            default:
                printf("Wrong option.\n");
        }
    }
    return 0;
}
```

Default values of function arguments

- In **C++** there exists a possibility to define a function with default values for arguments.
- It means that when calling such a function, we do not have to provide values for the arguments which already have the default values in the function definition.

```
float SUM3(float x, float y = 2.5, float z = 3.5)
{   return x + y + z;  }

float a, b, c;

SUM3( a, b, c ); // a + b + c
SUM3( a, b );    // a + b + 3.5
SUM3( a );       // a + 2.5 + 3.5
SUM3( );         // error, first argument is not default
SUM3( a, ,c );   // syntax error, two commas by each other
```

Recurrence

Theory and example

Functions – recurrence

- Recurrence happens when a definition of something uses itself.
- In programming language recurrence happens in functions.
- It means that a C function can call itself from the inside its instruction block.
- **Direct recurrence:**
  - In the body of function *P* we call function *P*:

```
void P(){
    /* ... */
    P();
    /* ... */
}
```

Functions – recurrence

- **Indirect recurrence:**

```
void P(){
     /* ... */
     Q(); // function Q calls for P
     /* ... */
}

void Q(){
     /* ... */
     P();
     /* ... */
}
```

Functions – recurrence

- **While creating a recurrence function, some important things must be considered:**
  - Calling a function by the same function means that the called function starts from the beginning with new values of parameters / arguments and new areas of memory are reserved for them.
  - Until called function ends, the calling instance of a function freezes and wait for the result of the calling.
- After calling for some function, we assume that when it finish its tasks, the program will continue its work. In other words: recursive calling will end at some moment.
- To make it possible, when calling a function a **return address** must be remembered, it usually contain the next instruction to execute after recursive ends.

Functions – recurrence

- Because called function can call another ones, it is necessary to store a sequence of returning addresses.
- While recursively called functions end their tasks, the returning addresses are used, in the reverse order as they were stored.
- The addresses are remembered in the so called ***stack*** of memory – a structure with a sequence of reads / writes given as **LIFO** sequence – Last In First Out.
- Such a mechanism is commonly used while functions calls each other, in particular in the recursive calling.
- In stack there are also stored: local function variables, its arguments, and depending on the situation also the returned values.

Functions – recurrence

Factorial

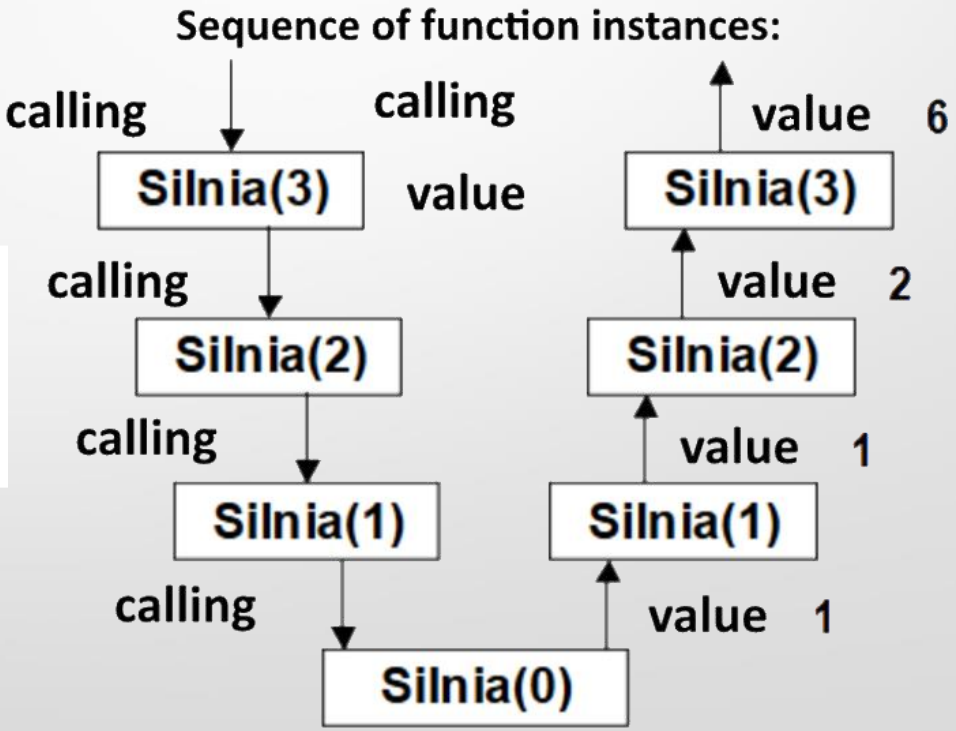
Problem decomposition:

$$S(0) = 1 \quad \Rightarrow \quad S(n) = 1 \quad \text{for} \quad n=0$$

$$S(n) = 1*2*3* \ldots *(n-1)*n$$

$$= S(n-1) * n \quad \Rightarrow \quad S(n) = S(n-1)*n \quad \text{for} \quad n>0$$

```
int Silnia(int n){
    if (n == 0) return 1;
    else return Silnia(n - 1) * n;
}
```



Functions – recurrence

- **Example: writing a number as a sequence of characters.**

```
#include <stdio.h>
void printd(int n) {
     if (n < 0) {
          putchar('-’);
          n = -n;
     }
     // digits are generated in reverse order; function printd first calls itself
     // to get the initial digits, finally to provide the last digit of a number
     if (n / 10)
          printd(n / 10);
     putchar(n % 10 + '0'); // % - modulo operator,
}
```

  - After calling *printd(123)* first call gets 123, second one is given (as argument) value 12, then the second one sends to the third one a value of 1.
  - Third called function writes '1', then program execution returns to the second function after calling the third one is finished, this second one writes '2' (12 % 10 = 2) then we return to the first calling of *printd*, which writes '3' (123 % 10 = 3).

Functions – recurrence

- Recurrence not necessarily spare memory, because we need a stack to store used values of arguments and results.
- Nor does it speeds up the function itself!
- Recurrence form is more compact and often easier to write and understand (depends…) than the iterative form of algorithm.
- Recursive functions are often used to do some work on the specific data structures like trees.
- It is **absolutely necessary** to provide a way for recurrence to end – it cannot be a sequence of infinite calling, because very fast we end up with an error of lack of memory (in stack).

Functions – recurrence

- **Example: write a recurrence form of a statement**

```
// multiply as long as n > 0
n  *  ( n - 2 )  *  ( n - 4 )  * ... //  > 0
```

- Recurrence version:

```
int IR ( int  n ) {
     return n <= 0 ? 1 : n * IR( n – 2 );
}
```

- Iterative version:

```
int IP ( int  n ) {
     int s = 1;
     while ( n > 0 ) {
          s *= n;
          n -= 2;
     }
     return s;
}
```

inline functions

- There are specific function with a keyword *inline* used just before the start of their definition / header.
- Such a code / function can be copy-pasted in memory each time such a function is being called – therefore its execution time is usually smaller than for the normal function.
- Keyword **inline** is being ignored if a function contains iterations.
- **Example:**

```
inline double WB(double x, double y, double z)
{
     return x > y ? z : x;
}

inline int multiply(int x, int y)
{
     return x*y;
}
```

Functions - Example

- Write a program storing surnames and ID for persons. All the data are stored in arrays. Key is a surname, value is ID. Use hash function.
- **Hash function**
  - Its main goal is to transform keys into indexes of a table.
  - In other words it generates a value based on the other value.
  - Given key K, in a first step one needs to compute corresponding index, in the next step check whether the location is free or already used to store something else.
  - A case where at a given index there is something already is called **collision.**
  - Algorithm providing alternative indexes is called collisions removing algorithm.

Functions – Example

```
/*
Hash coding
W – write new person (Surname, ID)
P – search for a person (by Surname) and show ID if found
Q – quit
*/

#include <stdio.h>
#include <string.h>

const int MAX = 100;

struct Person
{
     char *Surname;
     char ID[12];
};
```

Functions – Example

```
struct Position{
     struct Person *Who;
     int Busy;      // -1    : free,
                    // -2    : already taken and end of list,
                    //  >= 0 : next (person)|
};

int Hash(char *Name); // hash function
// finds free place in array
int Free(int poz, struct Position TAB[]);
// search by name
int Search(int poz, struct Position TAB[ ], char *Name);

int main( ){
     char Name[64];
     struct Person *NewOne;
     int index, number = 0;
     struct Position TAB[MAX]; // array of persons
     char Pol = 'X’;
     printf("\nProvide option [W, P, Q]\n");
```

Functions – Example

```
for (int i=0; i<MAX; ++i)
     TAB[i].Busy = -1; // initialization
do { // main loop
     printf("\n>"); // wait for options
     fflush(stdin);
     scanf("%c", &Pol);
     switch(Pol & 0x5F){
     case 'W' :
          if (number < MAX){ // adds new person
               number++; // number of person in table
               NewOne = new struct Person;
               //read person data
               printf("\nEnter name and ID: \n");
               scanf("%62s", Name);
               scanf("%11s", NewOne->ID);
               // assign memory for surname
               NewOne->Surname = new char[strlen(Name) + 1];
               strncpy(NewOne->Surname, Name, strlen(Name) + 1 );
               // put person to the table
               TAB[Free(Hash(Name), TAB)].Who = NewOne;
          } else
               printf("\nTable full!");
          break;
```

Functions – Example

```
        // search by name
        case 'P' :
              printf("\nEnter surname: \n");
              scanf("%63s", Name);
              // function returns array index or -1
              // if there is no such person
              index = Search(Hash(Name), TAB, Name);
              if (index == -1)
                    printf("\nNo such person.\n");
              else
                    printf("\n%s\n", TAB[index].Who->ID); // read ID
              break;
        case 'Q' :
              printf("\nQuit\n\n");
              break;
        default :
              printf("\nWrong option [W, P, Q]\n");
        }
  }
  while ((Pol & 0x5F) != 'Q')
        ; // until user writes Q or q
  return 0;
}
```

Functions – Example

```
// Simplest way is to add add values of letters
// So H("ala")=97+108+97=302
int Hash (char *Name){ // hash function
     int position = 0;
     while (*Name) // until end of text
          position += *Name++; // add all ASCII values of letters
     // to get proposed index of array for that person
     return position % MAX;
}

// finds free place in array to put a new person it
int Free (int pos, struct Position TAB[  ]){
     int end = pos, next;
     // Field busy stores information about elements on the list
     // which have the same hash value.
     if (TAB[pos].Busy == -1) {// is it free?
          TAB[pos].Busy = -2; // set to the end of list
          return pos;
     }
```

Functions – Example

```
else {
     // go to the end of list of object with the same hash value
     // at the end of it write the index of that person in main array

     while (TAB[end].Busy!= -2)
          end = TAB[end].Busy;
          // establish new index in array for that person
          next = (end + 1) % MAX;
          // if it is already busy, then search for new place with a step
          // of (next + 1) % MAX
          while (TAB[next].Busy != -1)
               next = (next + 1) % MAX;

          // set end-of-list sign
          TAB[next].Busy= -2;
          TAB[end].Busy = next; // add new element/index to the end of list
          return next;
     }
}
```

Functions – Example

```
// search by name
int Search(int pos, Position TAB[ ], char *Name){
     int next = pos;

     // there is no such person with a given surname
     if (TAB[next].Busy== -1) return -1;

     // person of a given surname has been found:
     if (strcmp(TAB[next].Who->Surname, Name) == 0)
          return next;
     else
          // we get the end of list (sign: -2) – person with a given surname
          // is not in the array
          if (TAB[next].Busy== -2)
               return -1;
          // continue searching the list in a recursive way
          else
               return Search(TAB[next].Busy, TAB, Name);
}
```

Questions?