Programming basics
(GKNB_INTA023)

Hatwagner F. Miklós, PhD.

Széchenyi István University, Győr, Hungary

https://github.com/sze-info/ProgrammingBasics

October 22, 2020

Functions

What is a *function*?

An identifiable and reusable block of the source code. Its behavior can be influenced by parameters.

Why do we use functions?

- Long source codes can be made more transparent and comprehesible by grouping the related lines of source code in functions (modularity)
- Functions can be reused (called, invoked) several times instead of copy and paste code snippets (decreasing code size)
  - They can be applied in one, specific program to avoid the repetition of code fractions
  - or even in multiple programs to avoid the repeated preparation of frequently used code snippets (eg. `sqrt`, `printf`)

Functions

Function *definition*

- Providing all formal information about the function: type of the return value, name (identifier), arguments (formal parameters: variables that are going to store parameter values given at function call), body of the function (inside curly braces, see eg. `main`).
- Functions can be defined exactly once.
- Definitions can be placed in source codes or in precompiled libraries.

absolute3.c

```c
double absolute(double number) {
  return number<0. ? -number : number;
}
```

Functions

Function call

- The function must be known at the place of call
- Passing control and (actual) parameters
- Call by *value*
- Returning program control and providing return value: `return`

absolute3.c

```c
int main(void) {
  double v;
  printf("Enter a number: "); scanf("%lf", &v);
  printf("Given number's absolute value: %f\n"
         "absolute(-3) == %f\nabsolute(v*3) == %f\n"
         "absolute(absolute(-3)) == %f\n",
         absolute(v), absolute(-3),
         absolute(v * 3), absolute(absolute(-3)));
  return 0;
}
```

Functions

Return value

- Return type cannot be an array
- Expression after `return`: *assignment* conversion (a kind of implicit type conversion) may be required
- `void`: expresses the lack of return value (“procedure”)

Formal parameters (arguments)

- No information about the number of arguments: `int main() {...}`
- No arguments: `int main(void) {...}`
- One parameter: `double absolute(double number) {...}`
- Two parameters:
  `double power(double base, double exponent) {...}`
- Actual parameters → *assignment* conversion → formal parameters
- Passing an array is a special case

Functions

The body of a function may contain everything that was allowed in the body of `main`, i.e.:

- Variable declarations
- References to items declared outside of the block
- Statements of activities

Returning from the function

- at the end of the function
- with a `return` statement (a function may contain several `return`-s)

search.c – Searching for the first occurrence of a character in a string

```c
int search(char haystack[], char needle) {
  unsigned i;
  for(i=0; haystack[i]!='\0'; i++) {
    if(haystack[i] == needle) return i;
  }
  return -1;
}
```

Functions

The definitions of functions cannot be embedded! (Except GCC, non-standard extension)

embedding.c

```c
#include <stdio.h>
int main(void) {
  double absolute(double number) {
    return number<0. ? -number : number;
  }
  printf("%f\n", absolute(-1.));
  return 0;
}
```

Compilation error (GCC: warning)

```
embedding.c: In function 'main':
embedding.c:3:3: warning: ISO C forbids nested functions [-Wpedantic]
double absolute(double number) {
```

Assignment conversion

Occurrences: when assigning a value to a variable, eg.

- converting the return value of a function

search.c `unsigned int` → `signed int`

```c
int search(char haystack[], char needle) {
  unsigned i;
  for(i=0; haystack[i]!='\0'; i++) {
    if(haystack[i] == needle) return i;
  }
  return -1;
}
```

Assignment conversion

Occurrences: when assigning a value to a variable, eg.

- when using the ?: operator

uppercase.c `int` → `char`

```c
int main(void) {
  char c;
  printf("Character: "); scanf("%c", &c);
  c = c>='a' and c<='z' ? c-'a'+'A' : c;
  printf("Uppercase shape: %c\n", c);
  return 0;
}
```

Assignment conversion

Occurrences: when assigning a value to a variable, eg.

- when converting the actual parameter of a function

absolute3.c `int` → `double`

```c
double absolute(double number) {
  return number<0. ? -number : number;
}
```

```c
            absolute(v), absolute(-3),
```

Assignment conversion

Details: C in a Nutshell

Some examples:

From	To	Outcome
signed+	unsigned	✓
signed−	unsigned	loss of sign
long int	int	danger of loss of value
int	double	danger of loss of precision
float	double	✓
double	float	danger of loss of precision
double	int	truncatenation of the fraction part

Function usage example

Services (functions) to be implemented:

Combination A *k-combination* of a set S is a subset of k distinct elements of S. If the set has n elements, the number of *k-combinations* is equal to

$$C_n^k = \frac{n!}{(n-k)!k!} = \binom{n}{k}$$

Example: given *three* fruits (say an apple, a pear, and a peach) how many combinations of *two* can be drawn from this set?

1. apple, pear
2. apple, peach
3. pear, peach

Function usage example

Services (functions) to be implemented:

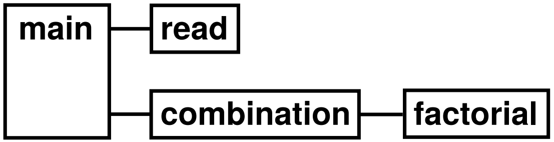
Factorial The factorial of a positive integer n, denoted by $n!$, is the product of all positive integers less than or equal to n.
$n! = \prod_{k=1}^{n} k$ for all $n \geq 0$ numbers.
$0! = 1$ according to convention.
Most basic use $\rightarrow$ Permutation: the number of possible distinct sequences of n distinct objects.
Example: how many distinct sequences of three distinct fruits (say an apple, a pear and a peach) can be created?

1. apple, pear, peach
2. apple, peach, pear
3. pear, apple, peach
4. pear, peach, apple
5. peach, apple, pear
6. peach, pear, apple

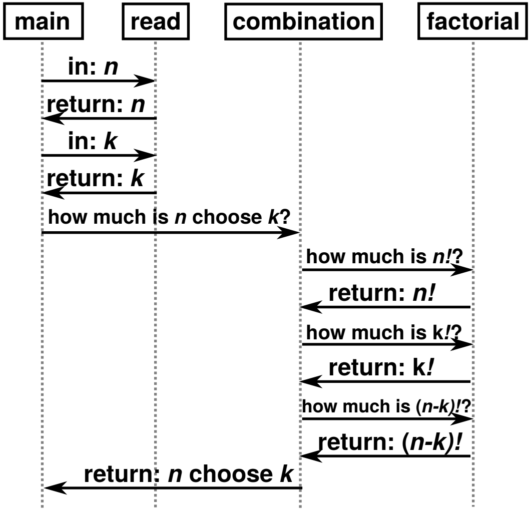
Function usage example

Services (functions) to be implemented:

Read	The value of n and k must be read
Main	Reading data, displaying $\binom{n}{k}$



Function usage example



Function usage example

nk1.c

```c
#include <stdio.h>
#include <limits.h>
#include <stdbool.h>
#include <iso646.h>

int read(int max) {
  int number;
  bool invalid;
  do {
    printf("Number: "); scanf("%d", &number);
    invalid = number<1 or number>max;
    if(invalid) printf("Invalid data!\n");
  } while(invalid);
  return number;
}
```

Function usage example

nk1.c

```c
unsigned long factorial(int n) {
  if(n < 2) return 1;
  unsigned long f = 1ul;
  for(int i=1; i<=n; i++) {
    f *= i;
  }
  return f;
}

unsigned long combination(int n, int k) {
  return factorial(n) / (factorial(k)*factorial(n-k));
}

int main(void) {
  int n = read(INT_MAX);
  int k = read(n);
  printf("%lu\n", combination(n, k));
  return 0;
}
```

Properties – lifetime

Lifetime (duration): *a period during runtime* when the variable/function exists, allocates memory. Types:

- Static
  - From the beginning of program execution to its end
  - All functions and *global* variables (declared outside functions) have static lifetime
  - Global variables are implicitely initialized: all bits are set to zero
  - Preferably the usage of global variables should be avoided
    - + Time of parameter-passing can be saved
    - – Hard to reuse code snippets, inflexible, environment-dependent code, danger of name conflicts, . . .

Properties – lifetime

- Local
  - Allocates memory from entering the block until leaving it
  - Function arguments, variables defined inside blocks (eg. in a block of an `if` statement) have local lifetime
  - Only explicit initialization is possible

nk1.c

```
unsigned long factorial(int n) {          17
  if(n < 2) return 1;                     18
  unsigned long f = 1ul;                  19
  for(int i=1; i<=n; i++) {               20
    f *= i;                               21
  }                                       22
  return f;                               23
}                                         24
```

Lifetimes (C99):

factorial exists during total program lifetime

n occupies memory 3x at the time of 3 function calls and frees memory at return (lines 18 and 23)

f occupies memory if *n* is great enough and cease at the moment of executing line 23

i occupies memory from reaching line 20 and cease when leaving the loop

Properties – scope

Scope: a portion of the source code where a name can be used to access its entity.

- Block (local)
  - From the declaration to the end of the block it was defined, including embedded blocks, too.
  - Eg. formal parameters of a function and its local variables
- File (global)
  - Functions and all identifiers declared outside of functions; from the point of declaration to the end of the file
  - Eg. functions, global variables

Properties – visibility

Visibility:

- The portion of the source code where the name can be legally accessed, referred.
- *Scope* and *visibility* usually coincide, though an entity may become temporarily hidden by the appearance of a duplicate name. Both entities exist but the *name* cannot be used to access the *original entity* until the scope of the duplicate name ends → creating duplicate names is a bad programming practice and should be avoided.

scopeVisibility.c

```c
int main(void) {
  int i = 0;             // int i is in scope and visible
  {                      // nested block
    double i = 2.14;// i is local name in the nested block
    i += 1;              // double i is in scope and visible;
                         // int i is also in scope but hidden
  }
                         // double i is out of scope
  i += 2;                // int i is visible and = 2
}
// int i and double i are both out of scope
```

Recursion

Recursive function call

- All functions are allowed to call themself directly or indirectly
- New memory areas are going to be reserved at every call for formal parameters and local variables
- Global variables remain at the same area
- Infinite recursion must be avoided!

nk2.c

```c
unsigned long factorial(int n) {
  if(n < 2) return 1;
  return n * factorial(n-1);
}
```

Recursion

power1.c Calculating power with multiplications

```c
long power(int base, unsigned exponent) {
  long result = 1;
  unsigned i;
  for(i=0; i<exponent; i++) {
    result *= base; }
  return result; }
```

power2.c Recursive power calculation, eg. $-3^5 = -3^{2^2} \times -3^1 = -243$

```c
long power(int base, unsigned exponent) {
  long result;
  if(exponent == 0) return 1;
  if(exponent == 1) return base;
  result = power(base, exponent/2);
  result *= result; // We don't invoke it twice!
  if(exponent%2 == 1) result *= base;
  return result; }
```

Recursion

Fibonacci sequence: each number is the sum of the two preceding ones, starting from 0 and 1. It counts the members of an imaginery rabbit family over time. How many pairs of rabbits will we have after n months if

- in the first month we only have 1 newborn rabbit-pair,
- newborn rabbit-pairs become fertile after 2 months,
- all fertile rabbit pairs give birth to another new pair of rabbits,
- and rabbits live forever :)

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

The beginning of the sequence is: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

Recursion

fibonacci1.c Iterative version

```c
unsigned long fibonacci(unsigned month) {
  unsigned long i=0, j=1, k;
  if(month < 2) return month;
  for(unsigned n=1; n<month; n++) {
    k = i+j;
    i = j;
    j = k;
  }
  return k;
}
```

fibonacci2.c Recursive version

```c
unsigned long fibonacci(unsigned month) {
  if(month < 2) return month;
  return fibonacci(month-1)+fibonacci(month-2);
}
```