Programming basics

(GKNB_INTA023)

Hatwagner F. Miklós, PhD.

Széchenyi István University, Győr, Hungary

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

October 15, 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`

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

```
          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 → 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