Basics of programming

Balázs Goldschmidt

Standard formatted input-output

Input

- Input
  - everything that provides information for the application
- Standard input
  - characters from the system
  - usually the keboard
  - might be file, pipe, etc.
  - application is oblivious about it

Output

- Output
  - anything that gets data from the application
- Standard output
  - output provided by the system
  - usually the screen
  - might be file, pipe, etc.
  - application is oblivious about it

Textual output

- Output is usually textual
  - characters
    - letters, digits, punctuation, whitespace, etc
- Must be converted from types
- Printf: formatted output
  - format string
  - decides how to convert types to text
  - undifined number of parameters

Printf format

- Syntax
  `printf(format, arg1, arg2, ...)`
- Number of arguments is only checked run-time
  - if wrong, error happens
- Format describes how args are to be treated
  - is a string, where % denotes the place of arguments
  - after % the actual conversion and format is specified
  - %[flags][size][.prec]type

Printf format (cont'd)

- %[flags][size][.prec]type
  - type
    - **d**ecimal, **u**nsigned, **o**ctal, he**x**adecimal, He**X**adecimal
    - *prefix:* **h**: short, **l**: long
    - **f**loat, **e**xponential
    - **c**har, **s**tring
    - % for percent sign
    - **g**: *e* or *f* whichever takes less space
    - **n**: number of chars printed, stored in int.

Printf format (cont'd)

- `%[flags][size][.prec]type`
  - flags
    - -, +, #, space, 0
  - size
    - min. size of the string
    - * specifies next arg should be used
  - prec
    - min number of digits for int
    - min number of decimal digits for 'e' and 'f'
    - number of significant digits for 'g'

Printf format example

- `printf("%d", 132)` → `132`
- `"num: %d", 132` → `num: 132`
- `"%04d", 132` → `0132`
- `"%03x", 255` → `0ff`
- `"%f", 13.45` → `13.450000`
- `"%05.1f", 13.45` → `013.4`
- `"%e", 13.45` → `1.345000e+01`
- `"%d %.2f", 3, 2.71` → `3 2.71`

Printf special chars

- Escape character
  - \
- Special characters
  - \t tabulator
  - \n new line
  - \" literal "
  - \0 NUL character (end of string)
  - \\ literal \

Textual input

- Input is usually textual
  - characters
    - letters, digits, punctuation, whitespace, etc
- Must be converted to types
- Scanf: formatted input
  - format string
  - decides how to convert text to types
  - undifined number of parameters

Scanf format

- `%[*][size]type`
  - type similar to printf
  - differences
    - **f**, e, *vs* **lf**, le *vs* **Lf**, Le: float, double, long double
    - s is for strings until whitespace
    - [...] for special char set
- any char in format string is literally matched
- parameters must have preceding &
- return value: number of successful reads

Scanf example

- Reading in an *int* and a *double* value
- Variable *n* stores the number of elements successfully read

```
int n, i;
double d;

n = scanf("%d %lf", &i, &d);
```


Functions

Functions

- In mathematics
  - f(x) = 2*x
  - f(x,y) = x*y/2
- In structured programming
  - used for structuring code
- Cannot be nested
  - all functions on same level
- Have
  - return type, name, arguments (parameters)

Structure of functions

```
double f(double x) {
   return 2*x;
}
```

- header
  - `type name (type1 arg1, type2 arg2, ...)`
  - arg1, arg2, etc are called formal parameters
- body
  - declaration of local variables
  - statements
  - *return* mandatory if return type is not void

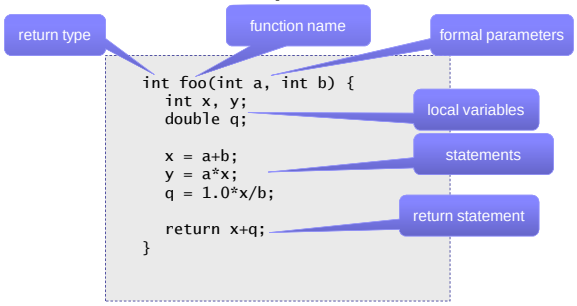
Calling functions

- Calling
  - `x = fun(exp1, exp2, ...)`
  - value of *exp1*, *exp2*, etc is assigned to formal parameters
  - value of expression in return statement is assigned to *x*
  - *exp1* and *exp2* are passed by value
    - modification is not shown on callers side
  - formal parameters behave similarly to local variables

Executing functions

- When called
  - stack is allocated for function
    - stores formal parameters and local variables
    - when function ends, stack is deallocated (freed)
  - actual parameters are copied to stack
  - execution is continuing in function
    - further functions might be called
  - function ends when
    - reaching return statement, return value is set
    - reaching end of body, no return value is set
- After finishing execution continues after call

Function example



Function example 2

- Function call and lifespan of local variables

```
int foo(int a, int b) {
   a = a/2;
   b = b/3;
   return a+b;
}

x = 3;
y = 10;
z = foo(x, y);
printf("%d, %d, %d\n", x, y, z);
/* -> 3, 10, 4 */
```


Function trivia

- Functions only communicate via
  - parameters
  - return value
- Print or read only when specified
- Unnecessary input-output results...
  - noisy code
  - unspecified behaviour
- Use functions instead of repeated code