Functions in C

Functions in C are more general than mathematical functions. They can be viewed as small programs on their own.

Functions have several advantages:

- Less code duplication – easier to read / update programs
- Simplifies debugging – each function can be verified separately
- Reusable code – the same functions can be used in different programs (e.g. printf)

Related functions are often collected in libraries like the C standard library, BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra PACKage).

Function definition

```
return-type function-name(parameters) {
    declarations
    statements
    return value;
}
```

- `return-type` type of value returned by function or `void` if none
- `function-name` unique name identifying function
- `parameters` comma-separated list of types and names of parameters
- `value` value returned upon termination (not needed if `return-type void`)

Function definition (cont.)

The list of parameters is a declaration on the form

```
type_1 par_1, ..., type_n par_n
```

and represents external values needed by the function. The list of parameters can be empty.

All declarations and statements that are valid in the main program can be used in the function definition, where they make up the function body.

Example, computing averages

```c
double average(double x, double y) {
    return (x+y)/2;
}
```

The `return` statement forces the function to return immediately, possibly before reaching the end of the function body.

Function declaration

Like variables a function must be declared before it can be used (called).

The declaration of a function resembles the definition,

```
return-type function-name(parameters types);
```

The function body is replaced by a semi-colon.

Parameters need not be named, it is sufficient to specify their types.

Example, declaring average

```c
double average(double, double);
```

Declaration is not necessary if the function definition precedes the first call.
Function calls

```
variable = function-name(arguments);
```

`variable` is assigned the return-value of the function.

The arguments are values with types corresponding to the function parameters.

For example,

```c
int main(int argc, char **argv) {
    double a = 1;
    double avg;

    avg = average(a, 3.0);
    return 0;
}
```

Implicit type conversion is used when the types of the arguments and parameters do not agree.

Function calls (cont.)

When a function is called the value of the argument is copied to the corresponding parameter. Changes made to the parameter inside the function will not affect the argument (call by value).

Example, the program

```c
#include <stdio.h>

void decrease(int i) {
    i--;    
    printf("%d ", i);
}

main() {
    int i = 1;
    printf("%d ", i);
    decrease(i);
    printf("%d\n", i);
}
```

gives the output 1 0 1.

Passing arguments by value or reference?

Transformation from polar to Cartesian coordinates: \((r, \varphi) \rightarrow (x, y)\).

Two input values & two output values.

C functions can only return one value:

- separate functions for computing \(x\) and \(y\)
- passing arguments by reference

Call by reference: any changes made to the parameter inside the function will affect the value of the argument outside.

We could pass the function \(r, \varphi\) by value and \(x, y\) by reference.

The subject will be covered tomorrow when we discuss `pointers`.

Variable scope

Each identifier is visible and can be used only in a certain context, or scope.

There are two kinds of scope in C: local and file scope.

File scope: Variables declared outside any function definition are visible to the end of the file.

These global variables should be avoided if possible to simplify debugging.

Local scope: Variables declared inside a block (compound statement) are visible to the end of the block.

Local scopes can be nested.

Scope rule: When a declaration names a visible identifier, the new declaration “hides” the old and the identifier takes on a different meaning.
## Variable scope (cont.)

Example,

```c
#include <stdio.h>
int i = 0; /* Declaration A */
int main() {
    printf("%d ", i); /* A */
    {
        int i = 1; /* Declaration B */
        printf("%d ", i); /* B */
        {
            int i = 2; /* Declaration C */
            printf("%d ", i); /* C */
            i++;
            printf("%d ", i); /* C */
        }
        printf("%d ", i); /* B */
    }
    printf("%d\n", i); /* A */
}
gives the output 0 1 2 3 1 0
```

## Storage classes

Local variables in C have automatic storage duration by default. Memory is allocated when the variable enters scope and is freed when the variable exits scope.

A local variable can be declared `static`, which means it is initialized only once and hence retains its value between function calls.

```
static type identifier;
```

A variable which is accessed and/or updated frequently can be declared as a register variable,

```
register type identifier;
```

This is a recommendation to the compiler to store the variable in a memory with less access time.

Storage classes for global variables and functions will not be covered in this course.

## Header files

C is a small language. Many of the common routines (`printf`, `malloc`, ...) are located in the C standard library.

The functions are declared in `header files` which are included by the preprocessor directive

```
#include <filename.h>
```

The list of directories searched for header files can be extended using the compiler option `-I`.

The directive

```
#include "filename.h"
```

searches only the current directory for header files.

For large programs it can be useful to write new header files.

Header files should contain only function declarations, variable declarations and preprocessor directives (Friday).

## Libraries

Header files only declare functions. The actual object code is often located in `libraries`, consisting of (pre-compiled) object files.

Libraries are needed when linking object files to obtain an executable.

The option `-l` name instructs the linker to look for the file `libname.a` in the directories searched for libraries.

The list of directories searched can be extended using the `-L` option.

One can create new libraries using the `ar` command.

A library should consist of related functions relying on as few external functions as possible.
An example - Newton’s method

Newton’s method for solving nonlinear equations \( f(x) = 0 \),

Until convergence: \( x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \)

File functions.c:

```c
#include <math.h>

double f(double x) {
    return (x-cos(x));
}

double fprime(double x) {
    return (1+sin(x));
}
```

File functions.h:

```c
double f(double);

double fprime(double);
```

File newton.c:

```c
#include <stdio.h>
#include <math.h>
#include "functions.h"

main() {
    double dx=1, x=0.5;
    int i=0;
    while (fabs(dx)>1e-12) {
        dx = -f(x) / fprime(x);
        x = x + dx;
        printf("%3d %17.12f %15e
",
                ++i, x, dx);
    }
}
```

Note: This program would not have converged if float had been used instead of double. (Tolerance must be \(\approx 5 \cdot 10^{-9}\) for convergence.)

An example - Newton’s method

Execution of example program:

```bash
> gcc -c functions.c
> gcc -c newton.c
> gcc functions.o newton.o -o newton -lm
> newton
```

```
1 0.755222417106  2.552224e-01
2 0.739141666150 -1.608075e-02
3 0.739085133921 -5.653223e-05
4 0.739085133215 -7.056461e-10
5 0.739085133215 -3.064197e-17
```

Note: This program would not have converged if float had been used instead of double. (Tolerance must be \(\approx 5 \cdot 10^{-9}\) for convergence.)

Convergence of Newton’s method

Relative floating point accuracy \( \equiv \min \{ \epsilon > 0 : (1+\epsilon) \neq 1 \} \).

Assume that Newton’s method is convergent to some \( x \neq 0 \). The program will converge as long as \( x + \frac{dx}{x} \neq x \), i.e. as long as

\[
1 + \frac{dx}{x} \\
\]

The relative accuracy of float is \(1.2 \cdot 10^{-7}\) which should be compared to \(2.2 \cdot 10^{-16}\) for double (IEEE).

(A thorough analysis should include other error sources as well.)