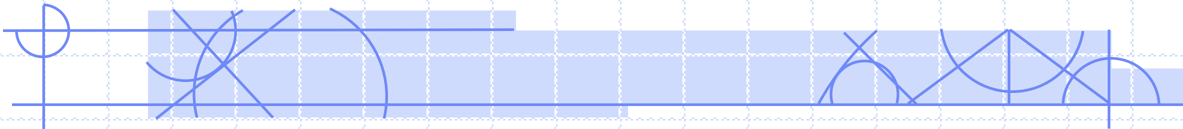
Functions



Modular Programming

- Breaking the solution into small, manageable program pieces – modules
- Modules are separately designed and sequentially executed
- Top-down design

Advantages

- Since modules are small parts of the solution, they are *simple* and *easy* to be created and tested.
- Modules design could be done *in parallel* for large projects.
- Once tested, modules are *reusable* in one or more programs and can be stored in libraries for further implementation.
- When a program is written in modules it is much more *readable* and understandable for people.
- When modules for computing values or performing single actions are available the programmer has the freedom to operate at higher *level of abstraction* while solving the problem.

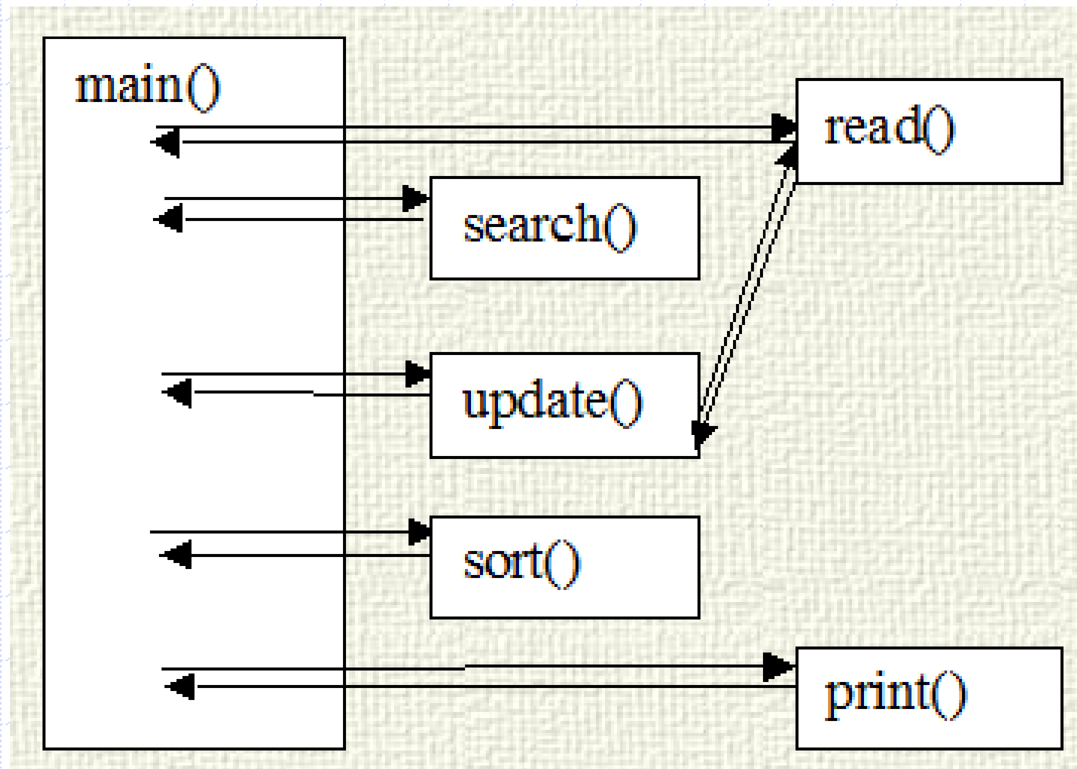
Modules in C Language

- Functions
  - relatively independent parts of the program
  - which have unique name and
  - can perform specific operations
- One program consists of one or more functions
- The first one is called main()

Functions

- The main() function calls, or invokes, the other functions
- The other functions can call each other, too
- A function may return one result to the calling function

Program Structure



Standard Library Functions

- Preliminary designed, developed and stored in standard libraries of the C environment
- These functions are called library functions
- When such a function is needed in a program it is just invoked by its name and specification of arguments
- The programmer doesn't have to take care about the function design and syntax.

Examples

- bx = sin(x)*sin(x);
- y = exp(x);
- z = pow(3.0, 2.0);
- ch = toupper(ch);

Implementation

- The programmer needs to know:
  - the location of the function, to include an appropriate header file;
  - the correct name, for identification;
  - the number and the type and the ordering of arguments, to send the adequate argument values;
  - the type of its returned result, to place it into an appropriate type variable or expression.

Implementation

- The argument can be given as an expression of the appropriate type
- The result of the function has to be assigned to a variable of appropriate type
- Examples:

```
x1 = (-b + sqrt(b * b - 4 * a * c))/(2 * a);
y = pow(cos(x), 2.0);
printf("%c", toupper(ch));
```

Programmer Defined Functions

- If a function doesn’t exist in any library
- A programmer designs it to
  - solve a small particular problem within the whole program
  - allow invocation by name and argument specification
  - release the programmer from care about details while designing the whole program structure

Example

- The function

$$f(x) = \frac{\sin(x)}{x}$$

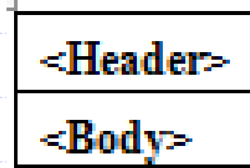
- can be designed as

```
double sinc(double x)
{
        if (fabs(x) < 0.001)
                return 1.0;
        else    return sin(x)/x;
}
```

- and invoked as `y = sinc(x);`

Function Definition

- One function definition consists of two parts:



- The header is the title line of the function. It contains an information that is important for the link between this function and other functions that call it.
- The body defines how the function works.

Function Header

- The **header** contains
  - the name of the function,
  - definitions of the function argument(s) and
  - the type of the result it returns

<Header>	`<result type> function_name(<arguments declarations>)`
<Body>	

- Examples

```
double pow(double x, double y) /* two arguments of type double */
int read()                     /* no arguments */
char key(void)                 /* no arguments */
```

Header

- The name of the function is a standard identifier chosen by the programmer
- The arguments declaration represent the information passed *to* the function when it is called to perform an action
- The <result type> is a standard data type of the value that is returned by the function
  - If the function gives no result, the <result type> is void
  - If the <result type> is not given, int is considered

Function Body

- Compound statement containing
  - declarations and
  - statements
  - enclosed in braces

<Header>	result_type function_name(<arguments declarations>)

<Body>	{
	 declarations
	 statements
	 }

Declarations

- Define some of the variables used inside the function
- The variables declared in the body of the function are called **local variables**
- Local variables can be used locally, by the *statements* inside this function
- The names of local variables are not known outside the function in which they are declared and their values are not valid anywhere else.

Statements

- Perform the operations which the function is designed for
- The operations are performed
  - on the function parameters,
  - on the local variables, as well as
  - on other external objects

The return Statement

- At least one of the *statements* in a function is a return statement
  - performs an action of returning the result to the routine which has invoked this function
- When a *return* statement is executed, it terminates the execution of the function and passes the control back to the calling statement
- The general form of this statement is:

  return <expression>;

- A function may have no return or more than one returns

Example

<Header>	`double sinc(double x)`
<Body>	`{ double eps=1.0E-04; if (fabs(x) < eps) return 1.0; else return sin(x)/x; }`

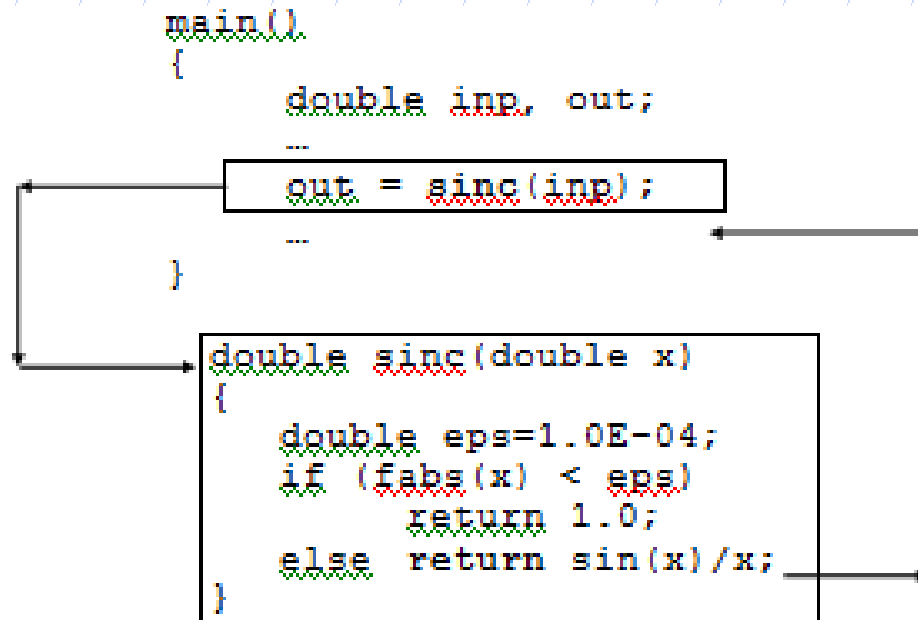
```
double sinc(double x)

{
    double eps=1.0E-04;
    if (fabs(x) < eps)
            return 1.0;
    else    return sin(x)/x;
}
```

- eps – local variable

Function Invokation

- Passing the control to the function



- Sending actual values to its parameters

Passing Parameters

- Actual parameters vs. formal arguments
  - The actual parameters match in **number, type** and **order** to the formal parameters
- Call-by-Value vs. Call-by-Reference

Actual Parameter *Formal Parameter*

inp [5.0] -> **x** [5.0]

Function Prototype

- Declares the function which will be called by another function and informs the compiler about the function name and parameters types
- can be placed
  - inside the calling function, with other variable declarations, as well
  - as a global declaration, with the preprocessor directives
- **header file** – contains function prototypes, can be referenced by an **include** statement

Function Prototype

- Similar to the header of the function
- Differs:
  - no names of arguments
  - semicolon at the end

```
main()
{
        double inp, out;
        double sinc(double);
        …
}
```

Example

```
#include <stdio.h>
int sqr(int);                    /* function prototype */

main()
{
  int b;
  scanf("%d", &b);
  printf("%d\n", sqr(b));       /* function invocation */
}

int sqr(int b)                   /* function definition */
{
   b = b * b;
   return b;
}
```

Global Variables

- Defined outside the function and still available for it
- Every function can access and change them – side effect

```
int radius;
main()
{
    ...
}
double area()
{
    return (3.1415*radius*radius);
}

double circum()
{
    return (2 * 3.1415 * radius);
}
```

Example

```
// Global and local variables
#include <stdio.h>
int n=20;                          /* a global variable n */
main()
{
       float vat(float);
       float value;                /* a local variable value, for main() only */
       printf("Value ");
       scanf("%f", &value);
       printf("The price with VAT is %.2f lv.\n", value+vat(value));
       printf("The price with VAT is %.2f lv.\n", (100.0+n)/100*value);
}

float vat(float x)
{
    int n = 30;                    /* a local variable n, for vat() only */
    return (x*n/100);
}
```

Macro Definitions

- Short functions defined in one line

  #define identifier(<parameter list>) <token_string>

- The preprocessor replaces every occurrence of identifier with the token string, except in quoted strings
- There may be zero or more parameters of the token string which are substituted for in later text

Example

```
#define SQ(x) ((x) * (x))
…
printf("%f %f", SQ(w),SQ(5+w));
…
```

- When *SQ(w)* occurs in the program it is replaced by *(w * w),* and *SQ(5+w)* is replaced by ((5+w)*(5+w))
- The identifier is always written in uppercase letters
- The parenthesis is placed just after the identifier
- No control of syntactic correctness

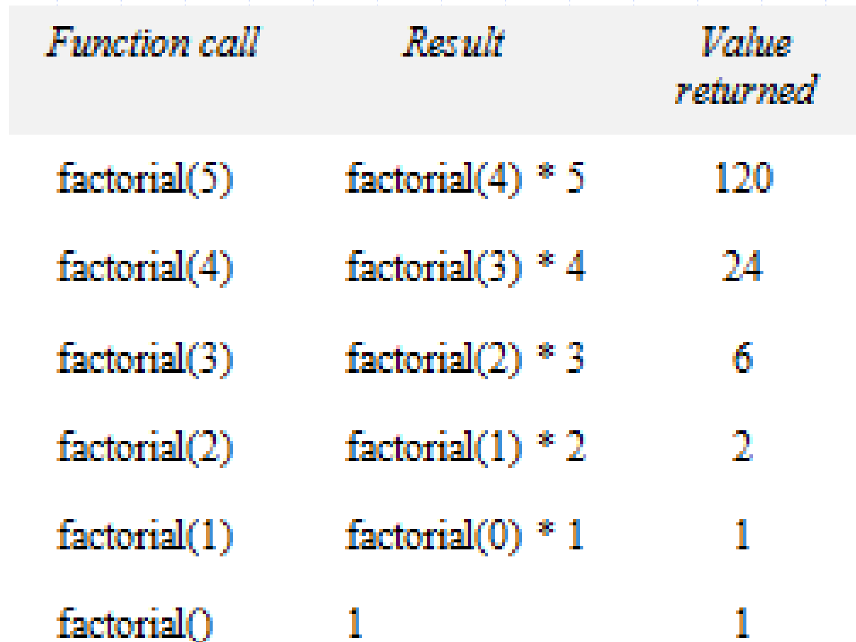
Recursion

- A function that calls itself
- In mathematics: the expression

  n! = 1.2.3.4.5 … n can be presented as
  - n(n-1)(n-2)(n-3)…3.2.1, for n >0, and
  - n!=1, for n=0
- In Programming: a recursive function

```
int factorial(int n)
{
   if (n <= 1) return 1;
   else return (n * factorial(n-1));
}
```

Recursion Iterations



Function call	*Result*	*Value returned*
factorial(5)	factorial(4) * 5	120
factorial(4)	factorial(3) * 4	24
factorial(3)	factorial(2) * 3	6
factorial(2)	factorial(1) * 2	2
factorial(1)	factorial(0) * 1	1
factorial()	1	1

Summary (1)

- C programs begin execution with a main() function, but they can contain other functions
- Any functions can refer to another function, defined in the same file or in another file, or in a library
- Some functions are pre-defined and stored into libraries, others have to be defined by the programmers.
- A programmer designed function is represented in a program by:
  1. function definition (description of the operations done by this function)
  2. function declaration, or prototype (introduction of the function)
  3. one or more function invocations, or calls (initializing the work of this function with particularly sent parameters to it)
- Generally, a function works with three types of variables:
  - variables sent to it by the function which calls it;
  - local variables;
  - global variables.

Summary (2)

- To accept and work with variables sent by the calling function, a function has a formal parameters list. It shows
  1. the number,
  2. the type and
  3. the order of function parameters which have to receive values from the calling function.
- The invocation statement contains an actual parameter list with the same number, type and order of values to be sent
- Variables declared inside a function are for a local use only - local variables. They have no meaning outside this function.
- Functions can operate with variables defined outside all functions - global variables
- Short functions can be described as macro definitions.
- Functions which call themselves are called recursive.

Key Terms

- module
- function
- function definition
- function declaration
- function prototype
- function call
- function invocation
- library function
- programmer-defined function
- formal parameter
- actual parameter
- local variable
- global variable
- void
- macro definitions
- recursive functions