1. **User-Defined Functions**: simplify programming task by distributing functionality among functions to make each one simpler.
   1. Abstraction to simplify programming.
      1. High-level description of a function.
      2. Encapsulates lower-level functions to be used by higher-level.
      3. Allows **top-down programming** to facilitate program design.
2. **Functions**: programs that carry out specific tasks; once written, it can accomplish tasks without needing an understanding of what goes on inside.
   1. Many functions in C are pre-defined and included in the C library.
      1. printf()
      2. scanf()
      3. Operators
   2. main()
      1. Always needed; no other function required.
      2. Processor begins and ends with it.
   3. 2 things we can do with functions
      1. **Define**: what they are to do.
         1. Functions cannot be defined within other functions.
            1. Before or after main()
         2. Requirements
            1. *Name*

Unique

Not reserved keyword in C

* + - * 1. *Type of value returned*

If any

* + - * 1. *Type and name of arguments* (inputs)
        2. *Declaration* *of automatic* (local) *variables*
        3. *Defining the body of the function*
        4. *Return a value of correct type*

If any

* + - 1. Method
         1. **Header**: first line of function definition.

Syntax

<return type> <function name>(<parameter list>)

<parameter list>: paired inputs **passed** to a function needed to run.

(…,<type> <name>, …)

Example

char comp\_grade(int grade)

void: put before the function name in the header to not return a value.

A function that does not accept any arguments also had void in its parameter definition.

Not essential.

Example

void funct1(void)

funct1()

return(expression of return type): value of expression defined is returned and function execution ends.

There can be multiple return statements in a function, but only one will be executed.

If function is not set to return anything, the function exits when the last statement is executed.

* + - * 1. **Body**: series of C statements, which describe the instructions to be executed on the variables defined.

As many as needed.

Inside {}

Variables

**Local**: automatic.

**Global**: data structures.

Parameters can be brought in as inputs.

* + - 1. Example
         1. float square(float x)
         2. {
         3. float y;
         4. y = x \* x;
         5. return y;
         6. }
    1. **Call**: ask them to do what they do.
       1. Process
          1. Functions can be called within other functions.
          2. Processor will stop what it is doing, and load **runtime stack**: list of instructions that the processor will execute.
          3. Returns final value and places it in place of the call.
          4. Then, processor returns to what it was doing.
       2. Functions can be called from within an arbitrary amount of **nesting**.
       3. Need to know information from **prototype**: describe the format of a function (name, inputs, outputs).
          1. **Name** of function

Not assigned by other functions or special reserved names

* + - * 1. **Pre-conditions**: arguments expected by function.

Inputs supplied to function

**Parameters**: when defining function.

**Arguments**: when calling function.

Within parentheses

* + - * 1. **Post-conditions**: what the function will do when it terminates.

Function can return a value, but it does not have to.

Many standard functions return something, but typically we ignore them.

* + - * 1. Only needed for user-defined functions
        2. Normally placed at the top of the program file after preprocessor directives.

Can be placed anywhere so long as they appear before their definition.

* + - * 1. Use is optional in C, but…

GCC compiler requires them.

C++ requires them.

* + - 1. **Call-by-value**: passing value of a variable as an argument to the function when called.
         1. The variable itself isn’t passed; only its value.
         2. C normally works this way.
      2. **Call-by-reference**: passing the **address** of the variable being referenced.
         1. Function can then change original variable.
         2. Need to know about pointers
      3. Examples
         1. double sqrt(double);

Accepts double

Returns double, replacing function call

* + - * 1. char comp\_grade(int);

Name: comp\_grade

Pre-condition: int

Post-condition: single character

* 1. Formal vs. Actual Parameters
     1. **Formal parameters**: listed in the function definition, acting as local variables within the function.
     2. **Actual parameters**: arguments; actual values passed in to the function when it is called.
     3. When calling a function…
        1. the number of arguments passed to it must be…
           1. the same number as the formal parameters in the function definition.
           2. of the same type.
        2. you do NOT put return or parameter types.

1. **Standard Functions (<math.h>)**
   1. Number of useful functions
   2. Algebra
      1. int abs(int);
         1. |int|
         2. Found in <stdlib.h>
      2. double fabs(double);
         1. |double|
      3. double pow(double, double);
         1. doubledouble
      4. double sqrt(double)
         1. Square root of double
   3. Trigonometry
      1. double sin(double)
      2. double cos(double)
      3. double tan(double)
2. Programming Style
   1. It is important to write code that humans can read and understand.
   2. Good habits make it easier for humans to read code.
      1. Indent properly
         1. Anything between {}s.
      2. Use plenty of whitespace
         1. a+b+c vs. a + b + c
      3. Give variables names that indicate their purpose
      4. Identify the meaning of a variable when first declared via comment on the same line or above, if its meaning is not obvious.
      5. Use UPPERCASE for constants
      6. Consistency with style
      7. Comment