1. **Function**: program module that should perform a specific task

Functions are predefined as part of the C++ library or defined by a programmer.

"Defined" means "coded".

2. **Output**  ←  **Program**  ←  **Input**

   Return Value  ←  **Function**  ←  Arguments or Parameters

   (1 value or no value)  (0 or more)

3. **Function Call**: causes the function to be executed; invokes the function

   **Syntax for call**: function name(argument list)

4. ```
#include <cmath>  // Preprocessor Directive
   // cmath contains the prototype for sqrt()

double x, y = 9.0, z = 1.0;

x = sqrt(25.0);

x = sqrt(y) + z;

cout << sqrt(x * y);
```

**Note**:
1. The return value is substituted for the function call.
2. An actual argument can be a constant, variable, or expression.
5. Function must be declared to compiler. 
   Reason: Compiler needs to know the return type and argument type.

   \texttt{double sqrt(double);} \quad \textit{// Function prototype}

   Better to include the header file than to code the prototype.

6. Definition of sqrt() is added to your executable program by the linker.

7. There is no exponentiation operator.
   \[
   x = y \\
   \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad // \texttt{double pow(double base, double exp);} \quad \textit{in} \texttt{cmath}
   \]
   \[
   \texttt{double x, y = 2.0;} \\
   \]
   \[
   x = \texttt{pow}(y, 5.0); \quad // \text{Sets x to y raised to the fifth power} \\
   \]
   \[
   x = \texttt{pow}(2.0, 4.0); \quad // 16.0 is assigned to x \\
   \]
   \[
   \texttt{int a = pow(2, 4);} \quad // \text{int arguments promoted to double for call} \\
   \quad \quad \quad \quad \quad \quad \quad // \texttt{pow(double (2), double (4))}
   \]
   
   Note: If base is negative, exponent must be integer

8. \[
   \quad // \texttt{int abs(int);} \quad \textit{in} \texttt{cstdlib}
   \]
   \[
   \texttt{cout << abs(5);} \quad // \text{outputs 5} \\
   \texttt{cout << abs(-5);} \quad // \text{outputs 5}
   \]
   
   \[
   \quad // \texttt{long labs(long);} \quad \textit{in} \texttt{cstdlib}
   \]
   \[
   \texttt{cout << labs(-40000);} \quad // \text{outputs 40000}
   \]
   
   \[
   \quad // \texttt{double fabs(double);} \quad \textit{in} \texttt{cmath}
   \]
   \[
   \texttt{cout << fabs(-40.5);} \quad // \text{outputs 40.5}
   \]

9. \[
   \quad // \texttt{double floor(double);} \quad \textit{in} \texttt{cmath}
   \]
   \[
   \texttt{cout << floor(3.8);} \quad // \text{outputs 3.0; rounds down} \\
   \]
   
   \[
   \quad // \texttt{double ceil(double);} \quad \textit{in} \texttt{cmath}
   \]
   \[
   \texttt{cout << ceil(3.2);} \quad // \text{outputs 4.0; rounds up}
   \]
10. **Black Box Analogy:**

electrical or mechanical device whose inner workings are hidden from view

User of a black box depends on *what* it does, not on *how* it does it;
does not know details of implementation.
// Sum function
#include <iostream>    // for cout
using namespace std;

    // Dummy Arguments Function Prototype
int sum_two_numbers(int operand1, int operand2);

int main()
{
    int a = 1, b = 2, c = 3, d = 4;

    // Actual Arguments Function Calls
    cout << "a+b=
    " << sum_two_numbers(a, b) << endl;  // Arguments are variables
    cout << "c+d=
    " << sum_two_numbers(c, d) << endl;
    cout << "9+8=
    " << sum_two_numbers(9, 8) << endl;  // Arguments are constants
    cout << "(a+1)+(2*b)=
    " << sum_two_numbers(a + 1, 2 * b) << endl; // Arguments are expressions
    return 0;
}

    // Formal Arguments Function Definition
int sum_two_numbers(int operand1, int operand2)
{
    int sum;
    sum = operand1 + operand2;
    return sum;
}

Sample Run:
    a + b = 3
    c + d = 7
    9 + 8 = 17
    (a + 1) + (2 * b) = 6

Term:
Structure Chart or Hierarchy Chart: Documentation that shows the relationships among the program modules (functions)
1. `sqrt()` and `pow()` are predefined functions.

Over 140 standard library functions

At times, you need to write your own.

**Programmer-defined function:** unit of code to perform a specific task; coded by the programmer

"defined" means "coded"

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Function Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td><code>sum_two_numbers</code></td>
<td>2 int numbers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>black box</td>
</tr>
</tbody>
</table>

2. **Function prototype:** Step 1 - Declare the function

```c
8 int sum_two_numbers(int operand1, int operand2);
```

**Declaration** informs the compiler that the program will use the function.

- a. type of return value: function can return 1 value or nothing
- b. function name:
  - Any valid identifier;
  - Should indicate task of function or return value
- c. type and number of arguments or parameters:
  - Multiple arguments are separated by commas.
  - Each argument has its own type.

```c
int sum_two_numbers(int operand1, operand2);
```

- d. ending semicolon terminates declaration statement

**Dummy Arguments:** `operand1` and `operand2`

The prototype allocates no memory.

Dummy arguments can be omitted,
but are recommended for documentation.

Dummy arguments do not have to be the same identifiers as the corresponding formal arguments.
3. **Function definition:**
(written for you with the library functions)

```c
int sum_two_numbers(int operand1, int operand2)
{
    int sum; // or
    sum = operand1 + operand2; // return operand1 + operand2;
    return sum;
}
```

a. **Function header** repeats info. given in prototype. No semicolon.

```c
int sum_two_numbers(int operand1, int operand2)
```

**Formal arguments:** operand1 and operand2

Do not have to be same as in prototype.

Creates a local variable known only to the function.

Difference from other local variables in that the formal arguments get their values from the corresponding actual arguments.

b. **Function body:** 1 or more statements in braces

Note: `main()` has same form:

```
int a = 1, b = 2, c = 3, d = 4;
...
<< sum_two_numbers(a, b) << endl;
...
<< sum_two_numbers(c, d) << endl;
...
return 0;
```
Actual arguments: a and b in line 16; then c and d in line 18

Actual arguments can be: constant, variable, or expression

5. **Stack**: data structure in which the last item added (pushed onto the stack; thinking of stack as a spring-loaded mechanism) is the first item removed (popped off of the stack); LIFO: Last In, First Out

Example: stack of coins or stack of dishes

**System Stack**: area of memory where copies of actual arguments and local variables are allocated when a function is called and deallocated when the function returns.

**Pass by Value**: function references a copy of the actual argument on the stack

**Local Variables**: variables that are declared within the body of a function

**Scope of a variable**: part of program where the variable can be used to access stored data.

6. Actual Argument checking at Compile Time:

When function is invoked, the compiler checks the type and number of actual arguments. If the number is incorrect, the call will not compile.

It will insert a cast if an actual argument is not the same type as corresponding formal argument.

Examples:

```cpp
int sum_two_numbers(int operand1, int operand2);

... 

cout << sum_two_numbers() << "\n" // Argument count is incorrect 
<< sum_two_numbers(a) << "\n" // Argument count is incorrect 
<< sum_two_numbers(a, b, c) << "\n" // Argument count is incorrect 
<< sum_two_numbers(75.5, 89.9); // expanded to 
    // sum_two_numbers(int(75.5), int(89.9));
```
// Compute average grade for one student - Version 4
#include <iostream> // for cin, cout
using namespace std;

// Function Prototypes
int compute_grade_sum(int score1, int score2, int score3);
double compute_grade_average(int score_sum);

int main()
{
    int grade1, grade2, grade3, grade_sum; // Local to main()
    double average_grade;
    char student_name[80]; // Local to main()

    // Input student's data
    cout << "Midterm Grade Report\n";
    cout << "Enter student's name ==> ";
    cin.getline(student_name, 80);
    cout << "Enter first grade =====> ";
    cin >> grade1;
    cout << "Enter second grade ====> ";
    cin >> grade2;
    cout << "Enter third grade =====> ";
    cin >> grade3;

    // Function Calls to calculate sum and average of the three grades
    grade_sum = compute_grade_sum(grade1, grade2, grade3);
    average_grade = compute_grade_average(grade_sum);

    // Output the student report
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(1);
    cout << "\nStudent's name: " << student_name;
    cout << "Test grades: " << grade1 << " " << grade2 << " " << grade3;
    cout << "\nGrade sum: " << grade_sum;
    cout << "\nGrade average: " << average_grade;
    cout << "\n\nEnd of Midterm Grade Report\n";
    return 0;
}

// Function Definitions
int compute_grade_sum(int score1, int score2, int score3)
{
    return score1 + score2 + score3;
}

double compute_grade_average(int score_sum)
{
    const double NUMBER_OF_SCORES = 3.0; // Local declared constant
    return score_sum / NUMBER_OF_SCORES;
}
Run 1:
Midterm Grade Report
Enter student's name ==> Carol Wong
Enter first grade =====> 100
Enter second grade =====> 76
Enter third grade =====> 83

Student's name: Carol Wong
Test grades: 100 76 83
Grade sum: 259
Grade average: 86.3

End of Midterm Grade Report

Run 2:
Midterm Grade Report
Enter student's name ==> Jim Garcia
Enter first grade =====> 100
Enter second grade =====> 100
Enter third grade =====> 100

Student's name: Jim Garcia
Test grades: 100 100 100
Grade sum: 300
Grade average: 100.0

End of Midterm Grade Report

Structure Chart:

```
main()
  \|-- compute_grade_sum()
  \|-- compute_grade_average()
```
Stack at return of `compute_grade_sum()` for Run 1:

<table>
<thead>
<tr>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>259</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>copy of grade1</th>
<th>score1</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>copy of grade2</th>
<th>score2</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>copy of grade3</th>
<th>score3</th>
</tr>
</thead>
<tbody>
<tr>
<td>83</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>student_name</th>
<th>average_grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carol Wong</td>
<td>garbage</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>grade_sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>garbage</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>grade3</th>
</tr>
</thead>
<tbody>
<tr>
<td>83</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>grade2</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>main()</th>
</tr>
</thead>
<tbody>
<tr>
<td>grade1</td>
</tr>
<tr>
<td>100</td>
</tr>
</tbody>
</table>

Stack at return of `compute_grade_average()` for Run 1:

<table>
<thead>
<tr>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>86.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>copy of grade_sum</th>
<th>score_sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>259</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>student_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carol Wong</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>average_grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>garbage</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>grade_sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>259</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>grade3</th>
</tr>
</thead>
<tbody>
<tr>
<td>83</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>grade2</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>main()</th>
</tr>
</thead>
<tbody>
<tr>
<td>grade1</td>
</tr>
<tr>
<td>100</td>
</tr>
</tbody>
</table>

Note:

1. Scope of identifiers
2. Why are dummy arguments dummy?
3. Placement of function definitions
// Compute average grade for one student - Version 5
#include <iostream> // for cin, cout
using namespace std;

// Function Prototypes
int grade_sum(int score1, int score2, int score3);
double average_grade(int score1, int score2, int score3);

int main()
{
    int grade1, grade2, grade3;
    char student_name[80];
    // Input student's data
    cout << "Midterm Grade Report\n";
    cout << "Enter student's name ==> ";
    cin.getline(student_name, 80);
    cout << "Enter first grade =====> ";
    cin >> grade1;
    cout << "Enter second grade =====> ";
    cin >> grade2;
    cout << "Enter third grade =====> ";
    cin >> grade3;

    // Output the student report
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(1);
    cout << "\nStudent's name: " << student_name;
    cout << "\nTest grades: "
         << grade1 << " " << grade2 << " " << grade3;
    cout << "\nGrade sum: " << grade_sum(grade1, grade2, grade3);
    cout << "\nGrade average: " << average_grade(grade1, grade2, grade3);
    cout << "\n\nEnd of Midterm Grade Report\n";
    return 0;
}

// Function Definitions
int grade_sum(int score1, int score2, int score3)
{
    return score1 + score2 + score3;
}

double average_grade(int score1, int score2, int score3)
{
    const double NUMBER_OF_SCORES = 3.0;
    return (score1 + score2 + score3) / NUMBER_OF_SCORES;
Run 1:
Midterm Grade Report
Enter student's name ==> Carol Wong
Enter first grade =====> 100
Enter second grade =====> 76
Enter third grade =====> 83

Student's name: Carol Wong
Test grades: 100 76 83
Grade sum: 259
Grade average: 86.3

End of Midterm Grade Report

Run 2:
Midterm Grade Report
Enter student's name ==> Jim Garcia
Enter first grade =====> 100
Enter second grade =====> 100
Enter third grade =====> 100

Student's name: Jim Garcia
Test grades: 100 100 100
Grade sum: 300
Grade average: 100.0

End of Midterm Grade Report

Structure Chart:

```
main()

grade_sum()  average_grade()
```
Stack at return of `grade_sum()`:

<table>
<thead>
<tr>
<th>copy of grade1</th>
<th>100</th>
<th>score1</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy of grade2</td>
<td>76</td>
<td>score2</td>
<td></td>
</tr>
<tr>
<td>copy of grade3</td>
<td>83</td>
<td>score3</td>
<td></td>
</tr>
<tr>
<td>student_name</td>
<td>Carol Wong</td>
<td></td>
<td></td>
</tr>
<tr>
<td>grade3</td>
<td>83</td>
<td></td>
<td></td>
</tr>
<tr>
<td>grade2</td>
<td>76</td>
<td></td>
<td></td>
</tr>
<tr>
<td>main()</td>
<td></td>
<td>grade1</td>
<td>100</td>
</tr>
</tbody>
</table>

Stack at return of `average_grade`:

<table>
<thead>
<tr>
<th>3.0</th>
<th>NUMBER_OF_SCORES</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy of grade1</td>
<td>100</td>
<td>score1</td>
</tr>
<tr>
<td>copy of grade2</td>
<td>76</td>
<td>score2</td>
</tr>
<tr>
<td>copy of grade3</td>
<td>83</td>
<td>score3</td>
</tr>
<tr>
<td>student_name</td>
<td>Carol Wong</td>
<td></td>
</tr>
<tr>
<td>grade3</td>
<td>83</td>
<td></td>
</tr>
<tr>
<td>grade2</td>
<td>76</td>
<td></td>
</tr>
<tr>
<td>main()</td>
<td></td>
<td>grade1</td>
</tr>
</tbody>
</table>
// Comparing Strings
// Usage of strcmp()
// Prototype: int strcmp(string1, string2);
// strcmp() returns zero if string1 and string2 are the same or
// a non-zero value if string1 and string2 are different
#include <iostream> // for cin, cout
#include <cstring> // for strcmp()
using namespace std;

int main()
{
    double prev_sal;
    char name[80];
    cout << "Enter first name (end to Quit):");
    cin.getline(name, 80); // Priming input
    while (strcmp(name, "end"); // while (strcmp(name, "end") != 0)
    {
        cout << "Enter salary: ");
        cin >> prev_sal;
        cout << "Name: " << name << " Salary: " << prev_sal << endl;
        cin.get(); // Discard \n before next cin.getline()
        cout << "Enter next name: ");
        cin.getline(name, 80); // Duplicate input
    }
    cout << "End of Program\n";
    return 0;
}

Sample Run:
Enter first name (end to Quit): Jose Bartolo
Enter salary: 100000
Name: Jose Bartolo Salary: 100000
Enter next name: Alice Kellenberger
Enter salary: 50000
Name: Alice Kellenberger Salary: 50000
Enter next name: Pat Sweeney
Enter salary: 76000.34
Name: Pat Sweeney Salary: 76000.34
Enter next name: END (not treated as Trailer data)
Enter salary: 200000
Name: END Salary: 200000
Enter next name: end
End of Program
Note:
line 21 in pseudocode:

while (name is Valid)
while (name is not trailer data)
while (name is not "end")
while (name != "end")
while (return value of strcmp() is non-zero)
1. Storing characters:

Each character is represented internally by its ASCII code.
(American Standard Code for Information Interchange)

8 bits for each character; 1 byte per character

<table>
<thead>
<tr>
<th>ASCII in base 10</th>
<th>Character</th>
<th>ASCII in base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>space</td>
<td>0010 0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>65</td>
<td>A</td>
<td>0100 0001</td>
</tr>
<tr>
<td>66</td>
<td>B</td>
<td>0100 0010</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>79</td>
<td>O</td>
<td>0100 1111</td>
</tr>
</tbody>
</table>

"BOB" stored as 0100 0010 0100 1111 0100 0010 0000 0000

null character

2. Collating Sequence: ranking or ordering of the characters base on their internal, ASCII representation

'.' < ',' < 'A' < 'B' < 'C' < ... < 'X' < 'Y' < 'Z'
32 < 44 < 65 < 66 < 67 < ... < 88 < 89 < 90

a. 'A' <= 'B'
   65 <= 66

b. 'C' < 'Z'
   67 < 90

c. 'D' >= ';
   68 >= 44
3. a. 83
   "SMITH"
   "WONG"
   87

   b. 73
   "ALICE"
   "ALLISON"
   76

   c. 0
   "ROB"
   "ROBERT"
   69

   d. 44
   "LE, EDWARD"
   "LEBOW, PEG"
   66

   e. 69
   "END"
   "end"
   101
// main() calls a function that returns a factorial

#include <iostream>    // for cin, cout
using namespace std;

int fact(int n);            // Function prototype

int main()
{
    int num;
    cout << "Factorial Computations\n";
    cout << "Enter an integer (negative to QUIT): ";
    cin >> num;
    while (num >= 0)
    {
        // Function call
        cout << "\t\t\t\t\t" << num << "! is " << fact(num);
        cout << "Enter another integer (negative to QUIT): ";
        cin >> num;
    }
    cout << "\nEnd of Factorial Program\n";
    return 0;
}

int fact(int n)            // Function definition
{
    int result = 1;
    while (n > 1)
    {
        result *= n;
        n--;
    }
    return result;
}

Sample Run:
Factorial Computations
Enter an integer (negative to QUIT): 0
Enter another integer (negative to QUIT): 1
Enter another integer (negative to QUIT): 2
Enter another integer (negative to QUIT): 3
Enter another integer (negative to QUIT): 4
Enter another integer (negative to QUIT): 5
Enter another integer (negative to QUIT): 6
Enter another integer (negative to QUIT): -1
End of Factorial Program
**Factorials:**

Factorials are not defined for negative numbers.

0! is 1

1! is 1 * 0!

2! is 2 * 1!

3! is 3 * 2!

4! is 4 * 3!

4 * 3 * 2!

4 * 3 * 2 * 1!

4 * 3 * 2 * 1 * 0!

4 * 3 * 2 * 1 * 1 which is 24

... 

n! is n * (n - 1)!

---

**Pictorial Representation of fact():**

```
n! \rightarrow \text{fact()} \rightarrow n
```

**Stack at beginning of fact():**

```
Stack  | 1   | result
       | copy of num | 4 | n  | fact()
       | main()      | num | 4 |
```

---

**Loop Cycle:**

1. Input number

2. Output its factorial