C Programming Note

What is C?

C is a general-purpose programming language created by Dennis Ritchie at the Bell Laboratories in 1972.

It is a very popular language, despite being old. The main reason for its popularity is because it is a fundamental language in the field of computer science.

C is strongly associated with UNIX, as it was developed to write the UNIX operating system.

Get Started With C

To start using C, you need two things:

- A text editor, like Notepad, to write C code
- A compiler, like GCC, to translate the C code into a language that the computer will understand

Syntax

```
#include <stdio.h>

int main() {
  printf("Hello World!");
  return 0;
}
```

Line 1: `#include <stdio.h>` is a **header file library** that lets us work with input and output functions, such as `printf()` (used in line 4). Header files add functionality to C programs.

Line 2: A blank line. C ignores white space. But we use it to make the code more readable.

Line 3: Another thing that always appear in a C program, is `main()`. This is called a **function**. Any code inside its curly brackets `{}` will be executed.

Line 4: `printf()` is a **function** used to output/print text to the screen. In our example it will output "Hello World!".

Line 5: `return 0` ends the `main()` function.

Line 6: Do not forget to add the closing curly bracket `}` to actually end the main function.

Note that: Every C statement ends with a semicolon `;`

Note: The body of `int main()` could also been written as:
`int main(){printf("Hello World!");return 0;}`

Remember: The compiler ignores white spaces. However, multiple lines makes the code more readable.

Output (Print Text)

To output values or print text in C, you can use the `printf()` function:

New Lines

To insert a new line, you can use the `\n` character:

Example

```
#include <stdio.h>

int main() {
  printf("Hello World!\n");
  printf("I am learning C.");
  printf("Hello World!\nI am learning C.\nAnd it is awesome!");
  return 0;
}
```

Comments in C

Comments can be used to explain code, and to make it more readable. It can also be used to prevent execution when testing alternative code.

Comments can be **singled-lined** or **multi-lined**.

Single-line Comments

Single-line comments start with two forward slashes (`//`).

Example

```
// This is a comment
printf("Hello World!");
```

C Multi-line Comments

Multi-line comments start with `/*` and ends with `*/`.

C Variables

Variables are containers for storing data values, like numbers and characters.

Declaring (Creating) Variables

To create a variable, specify the **type** and assign it a **value**:

Syntax

```
type variableName = value;
```

Where *type* is one of C types (such as `int`), and *variableName* is the name of the variable (such as **x** or **myName**). The **equal sign** is used to assign a value to the variable.

Example

Create a variable called **myNum** of type `int` and assign the value **15** to it:

```
int myNum = 15;
```

You can also declare a variable without assigning the value, and assign the value later:

Example

```
// Declare a variable
int myNum;

// Assign a value to the variable
myNum = 15;
```

Format Specifiers

Format specifiers are used together with the `printf()` function to tell the compiler what type of data the variable is storing. It is basically a placeholder for the variable value.

A format specifier starts with a percentage sign `%`, followed by a character.

Example

```
// Create variables
int myNum = 15;            // Integer (whole number)
float myFloatNum = 5.99;   // Floating point number
char myLetter = 'D';       // Character

// Print variables
printf("%d\n", myNum);
```

```c
printf("%f\n", myFloatNum);
printf("%c\n", myLetter);
```

Example

```c
int myNum = 15;
printf("My favorite number is: %d", myNum);
```

Change Variable Values

Note: If you assign a new value to an existing variable, it will overwrite the previous value:

```c
int myNum = 15;  // myNum is 15
myNum = 10;  // Now myNum is 10
```

You can also assign the value of one variable to another:

Example

```c
int myNum = 15;

int myOtherNum = 23;

// Assign the value of myOtherNum (23) to myNum
myNum = myOtherNum;

// myNum is now 23, instead of 15
printf("%d", myNum);
```

Add Variables Together

To add a variable to another variable, you can use the `+` operator:

Example

```c
int x = 5;
int y = 6;
int sum = x + y;
printf("%d", sum);
```

Declare Multiple Variables

To declare more than one variable of the same type, use a **comma-separated** list:

Example

```
int x = 5, y = 6, z = 50;
printf("%d", x + y + z);
```

You can also assign the **same value** to multiple variables of the same type:

Example

```
int x, y, z;
x = y = z = 50;
printf("%d", x + y + z);
```

C Variable Names

All C **variables** must be **identified** with **unique names**.

These unique names are called **identifiers**.

The **general rules** for naming variables are:

- Names can contain letters, digits and underscores
- Names must begin with a letter or an underscore (_)
- Names are case sensitive (`myVar` and `myvar` are different variables)
- Names cannot contain whitespaces or special characters like !, #, %, etc.
- Reserved words (such as `int`) cannot be used as names

Real-Life Example

```
// Student data
int studentID = 15;
int studentAge = 23;
float studentFee = 75.25;
char studentGrade = 'B';

// Print variables
printf("Student id: %d\n", studentID);
printf("Student age: %d\n", studentAge);
printf("Student fee: %f\n", studentFee);
printf("Student grade: %c", studentGrade);
```

C Data Types

The data type specifies the size and type of information the variable will store.

Data Type	Size	Description
`int`	2 or 4 bytes	Stores whole numbers, without decimals
`float`	4 bytes	Stores fractional numbers, containing one or more decimals. Sufficient for storing 6-7 decimal digits
`double`	8 bytes	Stores fractional numbers, containing one or more decimals. Sufficient for storing 15 decimal digits
`char`	1 byte	Stores a single character/letter/number, or ASCII values

Basic Format Specifiers

Format Specifier	Data Type
`%d` or `%i`	`int`
`%f`	`float`
`%lf`	`double`
`%c`	`char`

Type Conversion

Sometimes, you have to convert the value of one data type to another type. This is known as **type conversion**.

For example, if you try to divide two integers, `5` by `2`, you would expect the result to be `2.5`. But since we are working with integers (and not floating-point values), the following example will just output `2`

Example

```
int x = 5;
int y = 2;
int sum = 5 / 2;

printf("%d", sum); // Outputs 2
```

There are two types of conversion in C:

- **Implicit Conversion** (automatically)
- **Explicit Conversion** (manually)

Implicit Conversion

Implicit conversion is done automatically by the compiler when you assign a value of one type to another.

For example, if you assign an `int` value to a `float` type:

Example

```
// Automatic conversion: int to float
float myFloat = 9;

printf("%f", myFloat); // 9.000000
```

Explicit Conversion

Explicit conversion is done manually by placing the type in parentheses `()` in front of the value.

Considering our problem from the example above, we can now get the right result:

Example

```
// Manual conversion: int to float
float sum = (float) 5 / 2;

printf("%f", sum); // 2.500000
```

Constants

If you don't want others (or yourself) to change existing variable values, you can use the `const` keyword.

This will declare the variable as "constant", which means **unchangeable** and **read-only**:

Example

```
const int myNum = 15;  // myNum will always be 15
myNum = 10;  // error: assignment of read-only variable 'myNum'
```

Example

```
const int minutesPerHour = 60;
const float PI = 3.14;
```

Operators

Operators are used to perform operations on variables and values.

Example

```
int myNum = 100 + 50;
```

C divides the operators into the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Bitwise operators

Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations.

Operator	**Name**	**Description**	**Example**
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	++x
--	Decrement	Decreases the value of a variable by 1	--x

Assignment Operators

Assignment operators are used to assign values to variables.

Example

```
int x = 10;
```

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Comparison Operators

Comparison operators are used to compare two values (or variables). This is important in programming, because it helps us to find answers and make decisions.

The return value of a comparison is either `1` or `0`, which means **true** (`1`) or **false** (`0`).

Example

```
int x = 5;
int y = 3;
printf("%d", x > y); // returns 1 (true) because 5 is greater than 3
```

Operator	Name	Example
==	Equal to	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y

<=	Less than or equal to	x <= y

Logical Operators

You can also test for true or false values with logical operators.

Logical operators are used to determine the logic between variables or values:

Operator	Name	Description	Example
&&	Logical and	Returns true if both statements are true	x < 5 && x < 10
||	Logical or	Returns true if one of the statements is true	x < 5 || x < 4
!	Logical not	Reverse the result, returns false if the result is true	!(x < 5 && x < 10)

Sizeof Operator

The memory size (in bytes) of a data type or a variable can be found with the `sizeof` operator:

Example

```
int myInt;
float myFloat;
double myDouble;
char myChar;

printf("%lu\n", sizeof(myInt));
printf("%lu\n", sizeof(myFloat));
printf("%lu\n", sizeof(myDouble));
printf("%lu\n", sizeof(myChar));
```

C Booleans

C has a `bool` data type, which is known as **booleans**.

```
#include <stdbool.h>
```

A boolean variable is declared with the `bool` keyword and can only take the values `true` or `false`:

```
bool isProgrammingFun = true;
bool isFishTasty = false;
```

boolean values are returned as integers:

- `1` (or any other number that is not 0) represents `true`
- `0` represents `false`

Therefore, you must use the `%d` format specifier to print a boolean value:

Example

```
// Create boolean variables
bool isProgrammingFun = true;
bool isFishTasty = false;

// Return boolean values
printf("%d", isProgrammingFun);   // Returns 1 (true)
printf("%d", isFishTasty);        // Returns 0 (false)
```

Comparing Values and Variables

Example

```
printf("%d", 10 > 9);  // Returns 1 (true) because 10 is greater than 9
```

You can also compare two variables:

Example

```
int x = 10;
int y = 9;
printf("%d", x > y);
```

Real Life Example

In the example below, we use the `>=` comparison operator to find out if the age (`25`) is **greater than** OR **equal to** the voting age limit, which is set to `18`:

Example

```
int myAge = 25;
int votingAge = 18;

printf("%d", myAge >= votingAge); // Returns 1 (true), meaning 25 year olds are
allowed to vote!
```

C If ... Else

C has the following conditional statements:

- Use `if` to specify a block of code to be executed, if a specified condition is `true`
- Use `else` to specify a block of code to be executed, if the same condition is `false`
- Use `else if` to specify a new condition to test, if the first condition is `false`
- Use `switch` to specify many alternative blocks of code to be executed

The if Statement

Use the `if` statement to specify a block of code to be executed if a condition is `true`.

Syntax

```
if (condition) {
  // block of code to be executed if the condition is true
}
```

Example

```
if (20 > 18) {
  printf("20 is greater than 18");
}
```

Example

```
int x = 20;
int y = 18;
if (x > y) {
  printf("x is greater than y");
}
```

The else Statement

Use the `else` statement to specify a block of code to be executed if the condition is `false`.

Syntax

```
if (condition) {
  // block of code to be executed if the condition is true
} else {
  // block of code to be executed if the condition is false
}
```

Example

```
int time = 20;
if (time < 18) {
  printf("Good day.");
} else {
  printf("Good evening.");
}
// Outputs "Good evening."
```

The else if Statement

Use the `else if` statement to specify a new condition if the first condition is `false`.

Syntax

```
if (condition1) {
  // block of code to be executed if condition1 is true
} else if (condition2) {
  // block of code to be executed if the condition1 is false and condition2 is
true
} else {
  // block of code to be executed if the condition1 is false and condition2 is
false
}
```

Example

```
int time = 22;
if (time < 10) {
  printf("Good morning.");
} else if (time < 20) {
  printf("Good day.");
} else {
  printf("Good evening.");
}
// Outputs "Good evening."
```

This example shows how you can use `if..else` to find out if a number is positive or negative:

Example

```
int myNum = 10; // Is this a positive or negative number?

if (myNum > 0) {
  printf("The value is a positive number.");
} else if (myNum < 0) {
  printf("The value is a negative number.");
} else {
  printf("The value is 0.");
}
```

C Short Hand If Else

There is also a short-hand if else, which is known as the **ternary operator** because it consists of three operands.

Syntax

```
variable = (condition) ? expressionTrue : expressionFalse;
```

Instead of writing:

```
int time = 20;
if (time < 18) {
  printf("Good day.");
} else {
  printf("Good evening.");
}
```

You can simply write:

```
int time = 20;
(time < 18) ? printf("Good day.") : printf("Good evening.");
```

C Switch

Switch Statement

Instead of writing **many** `if..else` statements, you can use the `switch` statement.

The `switch` statement selects one of many code blocks to be executed:

Syntax

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

This is how it works:

- The `switch` expression is evaluated once
- The value of the expression is compared with the values of each `case`
- If there is a match, the associated block of code is executed
- The `break` statement breaks out of the switch block and stops the execution
- The `default` statement is optional, and specifies some code to run if there is no case match

Example

```
int day = 4;

switch (day) {
  case 1:
    printf("Monday");
    break;
  case 2:
    printf("Tuesday");
    break;
  case 3:
    printf("Wednesday");
    break;
  case 4:
    printf("Thursday");
    break;
  case 5:
    printf("Friday");
    break;
  case 6:
    printf("Saturday");
    break;
  case 7:
    printf("Sunday");
    break;
}

// Outputs "Thursday" (day 4)
```

The break Keyword

When C reaches a `break` keyword, it breaks out of the switch block.

This will stop the execution of more code and case testing inside the block.

When a match is found, and the job is done, it's time for a break. There is no need for more testing.

The default Keyword

The `default` keyword specifies some code to run if there is no case match:

C While Loop

Loops

Loops can execute a block of code as long as a specified condition is reached.

Loops are handy because they save time, reduce errors, and they make code more readable.

While Loop

The `while` loop loops through a block of code as long as a specified condition is `true`:

Syntax

```
while (condition) {
  // code block to be executed
}
```

Example

```
int i = 0;

while (i < 5) {
  printf("%d\n", i);
  i++;
}
```

The Do/While Loop

The `do/while` loop is a variant of the `while` loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

Syntax

```
do {
  // code block to be executed
}
while (condition);
```

Example

```
int i = 0;

do {
  printf("%d\n", i);
  i++;
}
while (i < 5);
```

C For Loop

For Loop

When you know exactly how many times you want to loop through a block of code, use the `for` loop instead of a `while` loop:

Syntax

```
for (statement 1; statement 2; statement 3) {
  // code block to be executed
}
```

Statement 1 is executed (one time) before the execution of the code block.

Statement 2 defines the condition for executing the code block.

Statement 3 is executed (every time) after the code block has been executed.

The example below will print the numbers 0 to 4:

Example

```
int i;

for (i = 0; i < 5; i++) {
  printf("%d\n", i);
}
```

Nested Loops

It is also possible to place a loop inside another loop. This is called a **nested loop**.

The "inner loop" will be executed one time for each iteration of the "outer loop":

Example

```
int i, j;

// Outer loop
for (i = 1; i <= 2; ++i) {
  printf("Outer: %d\n", i);  // Executes 2 times

  // Inner loop
  for (j = 1; j <= 3; ++j) {
    printf(" Inner: %d\n", j);  // Executes 6 times (2 * 3)
  }
}
```

C Break and Continue

Break

You have already seen the `break` statement used in an earlier chapter of this tutorial. It was used to "jump out" of a `switch` statement.

The `break` statement can also be used to jump out of a **loop**.

This example jumps out of the **for loop** when `i` is equal to 4:

```
int i;

for (i = 0; i < 10; i++) {
  if (i == 4) {
    break;
  }
  printf("%d\n", i);
}
```

Continue

The `continue` statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

This example skips the value of 4:

```
int i;

for (i = 0; i < 10; i++) {
  if (i == 4) {
    continue;
  }
  printf("%d\n", i);
}
```

Break and Continue in While Loop

You can also use `break` and `continue` in while loops:

Break Example

```
int i = 0;

while (i < 10) {
  if (i == 4) {
    break;
  }
  printf("%d\n", i);
  i++;
}
```

Continue Example

```
int i = 0;

while (i < 10) {
  if (i == 4) {
    i++;
    continue;
  }
  printf("%d\n", i);
  i++;
}
```

C Arrays

Arrays

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To create an array, define the data type (like `int`) and specify the name of the array followed by **square brackets []**.

To insert values to it, use a comma-separated list, inside curly braces:

```
int myNumbers[] = {25, 50, 75, 100};
```

Access the Elements of an Array

To access an array element, refer to its **index number**.

Array indexes start with **0**: [0] is the first element. [1] is the second element, etc.

This statement accesses the value of the **first element [0]** in `myNumbers`:

Example

```
int myNumbers[] = {25, 50, 75, 100};
printf("%d", myNumbers[0]);

// Outputs 25
```

Change an Array Element

To change the value of a specific element, refer to the index number:

Example

```
int myNumbers[] = {25, 50, 75, 100};
myNumbers[0] = 33;

printf("%d", myNumbers[0]);

// Now outputs 33 instead of 25
```

Loop Through an Array

You can loop through the array elements with the `for` loop.

The following example outputs all elements in the `myNumbers` array:

Example

```
int myNumbers[] = {25, 50, 75, 100};
int i;

for (i = 0; i < 4; i++) {
  printf("%d\n", myNumbers[i]);
}
```

Set Array Size

Another common way to create arrays, is to specify the size of the array, and add elements later:

Example

```
// Declare an array of four integers:
int myNumbers[4];

// Add elements
myNumbers[0] = 25;
myNumbers[1] = 50;
myNumbers[2] = 75;
myNumbers[3] = 100;
```

C Strings

Strings

Strings are used for storing text/characters.

For example, "Hello World" is a string of characters.

```
char greetings[] = "Hello World!";
```

Note that you have to use double quotes (`""`).

To output the string, you can use the `printf()` function together with the format specifier `%s` to tell C that we are now working with strings:

Example

```
char greetings[] = "Hello World!";
printf("%s", greetings);
```

Access Strings

Since strings are actually arrays in C, you can access a string by referring to its index number inside square brackets `[]`.

This example prints the **first character (0)** in **greetings**:

Example

```c
char greetings[] = "Hello World!";
printf("%c", greetings[0]);
```

Modify Strings

To change the value of a specific character in a string, refer to the index number, and use **single quotes**:

Example

```c
char greetings[] = "Hello World!";
greetings[0] = 'J';
printf("%s", greetings);
// Outputs Jello World! instead of Hello World!
```

Loop Through a String

You can also loop through the characters of a string, using a `for` loop:

Example

```c
char carName[] = "Volvo";
int i;

for (i = 0; i < 5; ++i) {
  printf("%c\n", carName[i]);
}
```

C Special Characters

Strings - Special Characters

Because strings must be written within quotes, C will misunderstand this string, and generate an error:

```c
char txt[] = "We are the so-called "Vikings" from the north.";
```

The backslash (`\`) escape character turns special characters into string characters:

Escape character	Result	Description
\'	'	Single quote

\"	"	Double quote
\\	\	Backslash

The sequence `\"` inserts a double quote in a string:

Example

```
char txt[] = "We are the so-called \"Vikings\" from the north.";
```

C User Input

User Input

You have already learned that `printf()` is used to **output values** in C.

To get **user input**, you can use the `scanf()` function:

Example

Output a number entered by the user:

```
// Create an integer variable that will store the number we get from the user
int myNum;

// Ask the user to type a number
printf("Type a number: \n");

// Get and save the number the user types
scanf("%d", &myNum);

// Output the number the user typed
printf("Your number is: %d", myNum);
```

The `scanf()` function takes two arguments: the format specifier of the variable (`%d` in the example above) and the reference operator (`&myNum`), which stores the memory address of the variable.

Multiple Inputs

The `scanf()` function also allow multiple inputs (an integer and a character in the following example):

Example

```
// Create an int and a char variable
int myNum;
char myChar;

// Ask the user to type a number AND a character
printf("Type a number AND a character and press enter: \n");

// Get and save the number AND character the user types
scanf("%d %c", &myNum, &myChar);

// Print the number
printf("Your number is: %d\n", myNum);

// Print the character
printf("Your character is: %c\n", myChar);
```

Take String Input

You can also get a string entered by the user:

Example

Output the name of a user:

```
// Create a string
char firstName[30];

// Ask the user to input some text
printf("Enter your first name: \n");

// Get and save the text
scanf("%s", firstName);

// Output the text
printf("Hello %s", firstName);
```

C Memory Address

Memory Address

When a variable is created in C, a memory address is assigned to the variable.

The memory address is the location of where the variable is stored on the computer.

When we assign a value to the variable, it is stored in this memory address.

To access it, use the reference operator (`&`), and the result represents where the variable is stored:

Example

```
int myAge = 43;
printf("%p", &myAge); // Outputs 0x7ffe5367e044
```

C Functions

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

Predefined Functions

So it turns out you already know what a function is. You have been using it the whole time while studying this tutorial!

For example, `main()` is a function, which is used to execute code, and `printf()` is a function; used to output/print text to the screen:

Example

```
int main() {
  printf("Hello World!");
  return 0;
}
```

Create a Function

To create (often referred to as ***declare***) your own function, specify the name of the function, followed by parentheses `()` and curly brackets `{}`:

Syntax

```
void myFunction() {
  // code to be executed
}
```

Example Explained

- `myFunction()` is the name of the function

- `void` means that the function does not have a return value. You will learn more about return values later in the next chapter
- Inside the function (the body), add code that defines what the function should do

Call a Function

Declared functions are not executed immediately. They are "saved for later use", and will be executed when they are called.

To call a function, write the function's name followed by two parentheses `()` and a semicolon `;`

In the following example, `myFunction()` is used to print a text (the action), when it is called:

Example

Inside `main`, call `myFunction()`:

```
// Create a function
void myFunction() {
  printf("I just got executed!");
}

int main() {
  myFunction(); // call the function
  return 0;
}

// Outputs "I just got executed!"
```

A function can be called multiple times:

Example

```
void myFunction() {
  printf("I just got executed!");
}

int main() {
  myFunction();
  myFunction();
  myFunction();
  return 0;
}

// I just got executed!
// I just got executed!
// I just got executed!
```

Return Values

Example

```c
int myFunction(int x) {
  return 5 + x;
}

int main() {
  printf("Result is: %d", myFunction(3));
  return 0;
}

// Outputs 8 (5 + 3)
```

This example returns the sum of a function with **two parameters**:

Example

```c
int myFunction(int x, int y) {
  return x + y;
}

int main() {
  printf("Result is: %d", myFunction(5, 3));
  return 0;
}

// Outputs 8 (5 + 3)
```

You can also store the result in a variable:

Example

```c
int myFunction(int x, int y) {
  return x + y;
}

int main() {
  int result = myFunction(5, 3);
  printf("Result is = %d", result);
  return 0;
}
// Outputs 8 (5 + 3)
```

Function Declaration and Definition

Example

```c
// Create a function
void myFunction() {
  printf("I just got executed!");
}

int main() {
  myFunction(); // call the function
  return 0;
}
```

A function consist of two parts:

- **Declaration:** the function's name, return type, and parameters (if any)
- **Definition:** the body of the function (code to be executed)

```c
void myFunction() { // declaration
  // the body of the function (definition)
}
```

For code optimization, it is recommended to separate the declaration and the definition of the function.

Example

```c
// Function declaration
void myFunction();

// The main method
int main() {
  myFunction();  // call the function
  return 0;
}

// Function definition
void myFunction() {
  printf("I just got executed!");
}
```

C Math Functions

Math Functions

There is also a list of math functions available, that allows you to perform mathematical tasks on numbers.

To use them, you must include the `math.h` **header file** in your program:

```
#include <math.h>
```

Square Root

To find the square root of a number, use the `sqrt()` function:

Example

```
printf("%f", sqrt(16));
```

Round a Number

The `ceil()` function rounds a number upwards to its nearest integer, and the `floor()` method rounds a number downwards to its nearest integer, and returns the result:

Example

```
printf("%f", ceil(1.4));
printf("%f", floor(1.4));
```

Power

The `pow()` function returns the value of ***x*** to the power of ***y*** (x^y):