

**COURSE OF DATA STRUCTURES AND ALGORITHMS WITH C++**

**Hours per week: 5**

**Instructors: HATANGIMBABAZI Hilaire AND HABANABASHAKA Jean Damascene**

1. **C++ PROGRAMMING CONCEPTS**

**I.1 Purpose statement of the Course**

At the end of this module you will be able to apply data structures in problem-solving scenarios and adeptly write C++ programs.You will learn to Apply C++ programming concepts, Analyze data structure and algorithms, Implement Linear Data Structure, Implement Non Linear Data Structure, Implement Hash Table and handle Data Retrieval, and Apply recursion for solving complex problems using C++.

Additionally, you will gain a comprehensive understanding of hash table implementations for efficient data retrieval and collision handling. With these competences, you will be empowered to tackle complex challenges independently, ensuring efficient and effective problem-solving in C++ programming.

**I.2 Overview of C++ programming**

**What is c++**

C++ is a cross-platform language that can be used to create high-performance applications.

C++ was developed by Bjarne Stroustrup at Bell labs in 1979, as an extension to the C language.

C++ gives programmers a high level of control over system resources and memory.

The language was updated 3 major times in 2011, 2014, and 2017 to C++11, C++14, and C++17.

**Why USE C++**

C++ is one of the world's most popular programming languages.

C++ can be found in today's operating systems, Graphical User Interfaces, and embedded systems.

C++ is an object oriented language which gives a clear structure to programs and allows code to be reused, lowering development costs.

**I.2.1 Evolution of C++**

Evolution of C++ can be traced back to 1980 when Bjarne Stroustrup developed a language he referred to as “C with Classes” at Bell Laboratories. Motivated object- oriented programming pioneered in Smalltalk, Stroustrup included powerful features into C with the design goal of supporting object-oriented programming while retaining backward compatibility with C.

By 1984, more enhancements had been added to “C with Classes” hence it was

renamed C++. Therefore, the name C++ uses C increment operator (++) to indicate that C++ is an enhancement of C. This integration of object orientation into procedural-oriented C makes C++ a multiparadigm language suitable for developing system software like operating systems.

**I.2.2 Features of C++**

The following are general features of C++ that makes it one of the most powerful and flexible programming supported by most computers.

1. **Portability:** Programs written in C++ are portable across multiple hardware and software platforms.
2. **Object-oriented programming:** The design goal of C++ is to support object- oriented programming. As mentioned earlier, instead of using functions that access global variables, both data and variables are encapsulated into an object.
3. **Keywords:** Keywords also referred to as **reserved words** are words that have special meaning in a language. C++has a large number such as include, main, while, for, if, else and return.
4. **Operators:** Operators are used to evaluate an expression that returns a value. This will be discussed in detail later.
5. **Storage in memory:** In C++ a variable is a named storage location in computer memory for holding data of a particular type.
6. **Identifiers:** In C++ programming, identifiers are symbolic names used to identify elements like variables and constants in a program. Because C++ s case sensitive, it is important to observe caution when creating user-defined identifiers.
7. **Case sensitive:** C++ is case-sensitive. Myname and MyName are different identifiers.
8. **Type checking:** C++ provides a rules and mechanism for checking data types before execution starts. If a compiler detects inconsistency, it ensures that the data conversions defined in the language or by the user do not cause runtime errors or system failure.

**I.3 Setup C++ environment**

Setting up a C++ development environment on both Windows and Linux involves installing a C++ compiler, an integrated development environment (IDE) or a text editor, and optionally, a build system. Here's a general overview of how to set up a C++ environment on both operating systems:

**I.3.1 Windows:**

**I.3.1.1 Install a C++ Compiler:**

For Windows, one of the most popular C++ compilers is Visual C++ from Microsoft, which comes with Visual Studio. You can download the free Visual Studio Community edition from the Microsoft website.

**I.3.1.2 Install an IDE (Optional):**

Visual Studio also serves as an IDE, providing a complete development environment. Alternatively, you can use Code::Blocks, Dev-C++, or CLion, which are popular C++ IDEs for Windows.

**Create a C++ Project:**

If you're using an IDE, create a new C++ project and start coding. If you're using a text editor, you can write your code in a plain text file with the ".cpp" extension.

**Compile and Run:**

In an IDE, you can build and run your code with the click of a button. If you're using a text editor, you'll need to use the command-line compiler. Visual C++ uses cl for compilation, while other compilers like GCC or MinGW may require different commands.

**I.3.2 Linux (Ubuntu-based):**

**I.3.2.1 Install a C++ Compiler:**

Most Linux distributions come with the GNU Compiler Collection (GCC) pre-installed. You can install it if it's not already available:

***sudo apt-get update***

***sudo apt-get install g++***

**I.3.2.2 Install an IDE (Optional):**

* You can use command-line tools like g++ for compiling and a text editor (e.g: gedit, Nano, Vim, or Emacs) for writing code.
* For an IDE, you can install Code::Blocks, CLion, or use Visual Studio Code with C/C++ extensions.

**Create a C++ File:**

Use your preferred text editor to create a C++ source file (e.g., myprogram.cpp).

Write and Save Code:

Write your C++ code in the source file and save it.

**Compile and Run:**

* Open a terminal and navigate to the directory containing your source file.
* Compile the code using g++:

***g++ myprogram.cpp -o myprogram***

* Run the executable:

***./myprogram***

These are the general steps for setting up a C++ development environment on Windows and Linux. The specific steps may vary depending on your chosen tools and distribution, but this should give you a solid foundation to start coding in C++.

**I.4 C++ first program**

Let's create a simple "Hello, World!" program in C++. This program will display the text "Hello, World!" on the screen. Here's the code:

***#include <iostream>***

***using namespace std;***

***int main()***

***{***

***cout << "Hello World! ";***

***return 0;***

***}***

**Note: A C program is a C++ program**

***#include <stdio.h>***

***int main()***

***{***

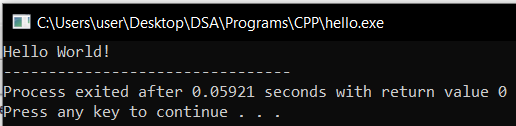
***printf( "Hello World! ");***

***return 0;***

***}***

**When saved as hello.cpp, the program will compile**

**Output:**

****

**Program explained:**

Line 1: #include <iostream>:This line includes the input/output stream library, which allows you to work with input and output in C++.

Line 2: using namespace std means that we can use names for objects and variables from the standard library. using namespace std; namespace is a feature in C++ used to ensure that identifiers do not overlap due to naming conflict. Identifiers may overlap by sharing different parts of a program. Each namespace such as std (standard) defines a scope in which identifiers are placed.

Don't worry if you don't understand how using namespace std works. Just think of it as something that (almost) always appears in your program.

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

Line 4: cout (pronounced "see-out") stands for ***console out*** is an object used to output/print text. In our example it will output "Hello World". It uses the **insertion operator (<<)**

You can add as many cout objects as you want. However, note that it does not insert a new line at the end of the output **if no new-line statement is used**.

Note: Every C++ statement ends with a semicolon ;.

Note: The body of int main() could also been written as:

int main () { cout << "Hello World! "; return 0; }

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

Line 5: return 0 ends the main function.

**New Lines**

To insert a new line, you can use the \n character. Two \n characters after each other will create a blank line:

#include <iostream>

using namespace std;

int main() {

cout << "Hello World! \n";//New line

cout << "Hello World again! \n\n";// Blank line

cout << "I am learning C++";

return 0;

}

Another way to insert a new line, is with the endl manipulator:

#include <iostream>

using namespace std;

int main() {

cout << "Hello World!" << endl;

cout << "I am learning C++";

return 0;

}

**I.5. C++ input stream(User input)**

In C++ handling input is done using the cin combined with **>>** known as stream extraction operator. cin represents the standard input device (or **C**onsole **IN**put),

i.e., keyboard. The symbol **>>** after the cin. The **>>** operator causes data input such as an integer value to be input from cin into computer memory.

You have already learned that cout is used to output (print) values. Now we will use cin to get user input.

cin is a predefined variable that reads data from the keyboard with the **extraction operator** (>>).

In the following example, the user can input a number, which is stored in the variable x. Then we print the value of x:

**Example**

int x;

cout << "Type a number: "; // Type a number and press enter

cin >> x; // Get user input from the keyboard

cout << "Your number is: " << x; // Display the input value

cout is pronounced "see-out". Used for output, and uses the insertion operator (<<)

cin is pronounced "see-in". Used for input, and uses the extraction operator (>>)

**I.6. Data types in C++**

The computer memory is organized into cells that can store one or more bytes. A byte is the minimum amount of memory that we can manage in C++. To declare a variable, you must declare the type of variable so that the computer reserves enough bytes to store a value of that type.

**Primitive types:** int, float, double, boolean and char

**Complex data types**

A complex data type is a combination of data of similar or different types. C++ supports complex data types such as **string, array, struct (record), enumerated type, linked lists** and **pointers**. Apart from string data type, other examples of complex data types include arrays, linked lists, stacks, queues, trees and graphs.

**C++ strings**

Strings are used for storing text.

A string variable contains a collection of characters surrounded by double quotes: eg. string greeting = "Hello";

To use strings, you must include an additional header file in the source code, the <string> library:

String Concatenation

The + operator can be used between strings to add them together to make a new string. This is called concatenation:

**Example**

string firstName = "John ";

string lastName = "Doe";

string fullName = firstName + lastName;

cout << fullName;

**String length**

A string in C++ is actually an object, which contains functions that can perform certain operations on strings. For example, the length of a string can be found with the length() function:

string txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

cout << "The length of the txt string is: " << txt.length()

**Access Strings**

You can access the characters in a string by referring to its index number inside square brackets [].

This example prints the first character in myString:

string myString = "Hello";

cout << myString[0];

// Outputs H

**Change String Characters**

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

**Example**

string myString = "Hello";

myString[0] = 'J';

cout << myString;

// Outputs Jello instead of Hello

**User Input Strings**

It is possible to use the extraction operator >> on cin to display a string entered by a user:

**Example**

string firstName;

cout << "Type your first name: ";

cin >> firstName; // get user input from the keyboard

cout << "Your name is: " << firstName;

// Type your first name: John

// Your name is: John

However, cin considers a space (whitespace, tabs, etc) as a terminating character, which means that it can only display a single word (even if you type many words):

string fullName;

cout << "Type your full name: ";

cin >> fullName;

cout << "Your name is: " << fullName;

// Type your full name: John Doe

// Your name is: John

From the example above, you would expect the program to print "John Doe", but it only prints "John".

**getline**

That's why, when working with strings, we often use the getline() function to read a line of text. It takes cin as the first parameter, and the string variable as second:

string fullName;

cout << "Type your full name: ";

getline (cin, fullName);

cout << "Your name is: " << fullName;

// Type your full name: John Doe

// Your name is: John Doe

**Note:** When using getline() after some inputs, the function will fail to read inputs because when cin.getline() reads from the input, there is a newline character left in the input stream, so it doesn't read your c-string. Use cin.ignore() before calling getline().

cout<<"Enter age";

cin>>age;

cout<<"Enter fullname:\t";

**cin.ignore();**

**getline(cin,fullname);**

**Exercises**

1. **Write a program that prompts the user to enter the firstName,lastName,age and display all information in one line.**
2. **Write a program which asks the user to enter**

* **fullname**
* **initial amount,**
* **the interest rate per year,**
* **payment time in terms of years**

**And finally Compute the total interest for the user. The program should then display all data on different lines.**

**I.7. C++ identifiers**

The names of variables, functions, labels, and various other user-defined **objects are called identifiers.**

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

These unique names are called identifiers.

Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).

Note: It is recommended to use descriptive names in order to create understandable and maintainable code.

The general rules for constructing names for variables (unique identifiers) 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 (like C++ keywords, such as int) cannot be used as names

**I.8. Keywords in C++**

C++ has reserved words, which have a unique meaning and must not be used for any other purposes. The reserved words already used are **main, int , return**, and **using**. See other examples below:

else, new,this, auto, enum, bool, true, private, try, public, protected, case, false, true, break, class, for, return,const, signed, unsigned, void, static, int, float, continue, if, break, switch, while, etc…..

**I.9. Namespace**

In C++, a namespace is a mechanism that allows you to organize and group code elements, such as variables, functions, and classes, to prevent naming conflicts and provide better modularity and organization within your code. Namespaces help avoid naming collisions between identifiers in different parts of your program, especially when working with libraries or multiple developers.

* Namespace is a feature added in C++ and not present in C.
* A namespace is a declarative region that provides a scope to the identifiers (names of the types, function, variables etc) inside it.
* Multiple namespace blocks with the same name are allowed. All declarations within those blocks are declared in the named scope.

Here's how you can define and use namespaces in C++:

**Defining a Namespace:** You define a namespace using the namespace keyword followed by the namespace's name. Typically, this is done in a separate header or source file.

Consider the following C++ code.

// A program to demonstrate need of namespace

**int** main()

{

**int** value;

value = 0;

**double** value; // Error here

value = 0.0;

}

Compiler Error:

'value' has a previous declaration as 'int value'

In each scope, a name can only represent one entity. So, there cannot be two variables with the same name in the same scope. Using namespaces, we can create two variables or member functions having the same name.

Namespace definition syntax:

***namespace namespace\_name***

***{***

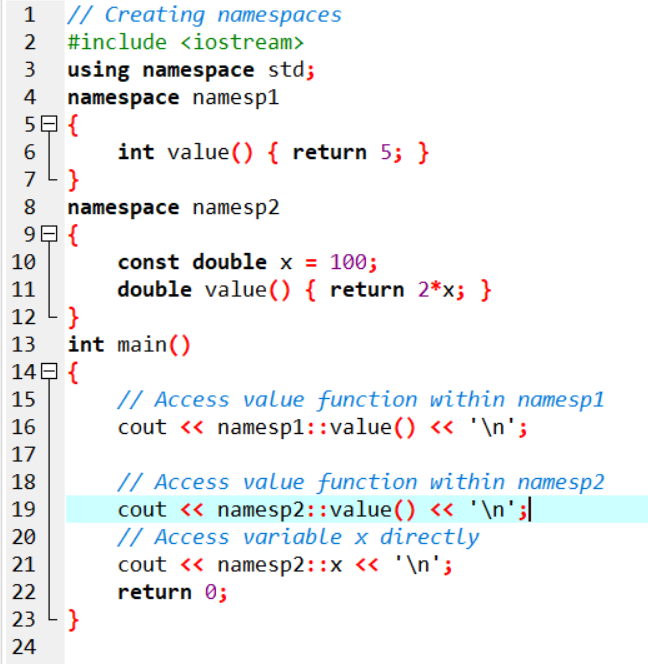
***// c++ code in namespace\_name's scope are put here***

***}***

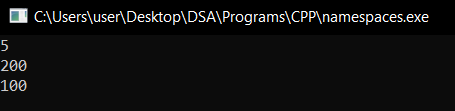
**Namespace’s Characteristics:**

* Namespace declarations can be nested within another namespace.
* Namespace declarations don’t have access specifiers. (Public or private)
* No need to give a semicolon after the closing brace of the definition of namespace.
* We can split the definition of namespace over several units.

**Example:**

******

**Output:**

****

5: Value returned by **value(**) in the namespace **namesp1**

200: Value returned by **value(**) in the namespace **namesp2**

100: Value constant **x** in the namespace **namesp2**

**Exercise**

1. Create a program with a namespace userDefined with a variable insideNamespace of type integer and give it initial value.
   1. Create a function cout inside the namespace which returns an integer as the value of insideNamespace.
   2. Create a global variable myGlobal of type integer and give it initial value
   3. Create a global function cout and return the square of myGlobal
   4. Create the main function with integer local variable cout with initial value.
   5. Display all variables defined above and display the output of cout() function in userDefined namespace and cout() global function using the following description:
      1. <<"The local variable cout in main is"<<
      2. <<"The variable in userDefined namespace is"<<
      3. <<"The output of cout() in userDefined is"<<
      4. <<"The value of myGlobal is"<<
      5. <<" The output of global cout() is"<<

**Omitting namespace**

You might see some C++ programs that run without the standard namespace library. The using namespace std line can be omitted and replaced with the std keyword, followed by the :: operator for some objects:

Example

#include <iostream>

int main() {

std::cout << "Hello World!";

return 0;

}

**I.10. C++ constants**

In C++, constants may be classified into **literal constants and symbolic constants.**

1. **Literal Constants**

Literals constants are used to express particular values within a program. For example, in the following statement, 25 is a literal constant because you can neither assign another value to it nor can you change it.

x + **25**;

Literal constants can be classified into integer numerals, floating-point numerals,characters, strings and boolean constants.

1. **Symbolic Constants**

A symbolic constant is a constant that is represented using a symbolic name. Once

a symbolic constant is initialized, its value cannot be changed. There are two ways

to declare a symbolic constant in C++ are:

1. Using preprocessor directive **#define**. Following is the form to use #define preprocessor to define a constant −

#define identifier value

For example, the following statement declares a symbolic constant named **numberOfDistricts** that is replaced by 30 during Execution:

**#define numberOfDistricts 30;**

1. Using keyword **const** followed by the **data type** of the symbolic constant as shown in the statement below:

For example,

**const short int numberOfDistrict = 30;**

The advantage is that the compiler is able to determine data type of the constant hence preventing possible runtime errors

**Exercise**

Build a c++ program that compute the circumference and area of the circle

1. Declare a double constant which contains PI=3.14159265
2. Request the user to enter the radius of a circle and compute the area and circumference of the circle.

**Output formatting**

In programming, creating nicely formatted output is a good programming practice to improve readability of output and the user interface. In C++, the output stream has special characters and objects called **manipulators** in <iostream> used to format numbers, characters and strings.

1. **The endl manipulator**

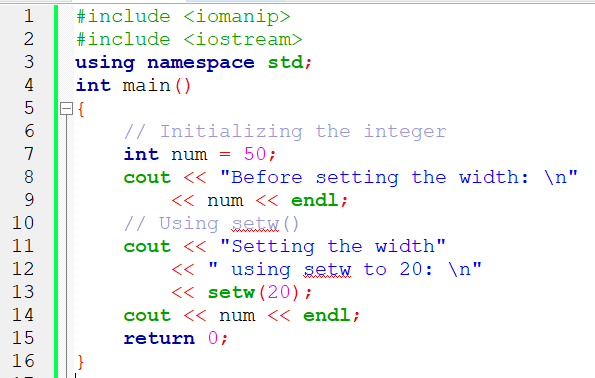
When supplied with operator << at the end of a statements, endl object causes a newline

character to be inserted at the end of a line.

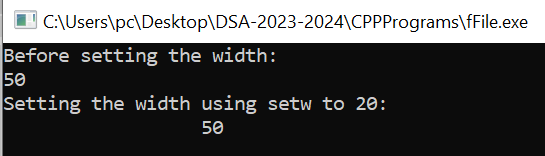
**cout << “Hello, world!” << endl;**

1. **The setw() manipulator**

To produce number and string output formatted to fixed width in terms of number of characters, C++ has a manipulator object called setw() inside the <iomanip> library.. For example, setw(20) in the statement below adjusts the value of num. If the characters are fewer, a blank space is inserted on the left of the output.



**Output**

****

1. setw() is a library function in C++.
2. setw() is defined in the **iomanip** library.
3. setw() will set field width.

setw() sets the number of characters to be used as the field width for the next insertion operation.

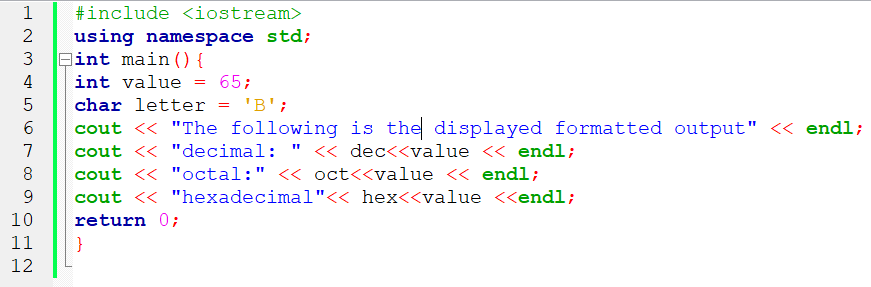
c) **The setprecision() manipulator**

In C++, formatting floating point numbers may be rounded off to the nearest integer using **setprecision()** manipulator. The object is used together with fixed or scientific manipulators to specify the number of digits to be displayed.

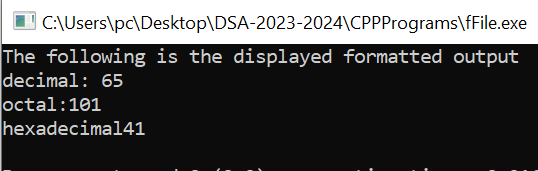
**d) Format Base of Integer Output**

In computing, the commonly used numeric constants are decimal integers, floating point (real numbers), octal (base 8) and hexadecimal (base 16). In C++ we use format specifiers to format or convert a number from one base to another. To change the base of printed values use dec , oct , and hex manipulators. The following program demonstrates how to format the three number systems:

Example:



Output:

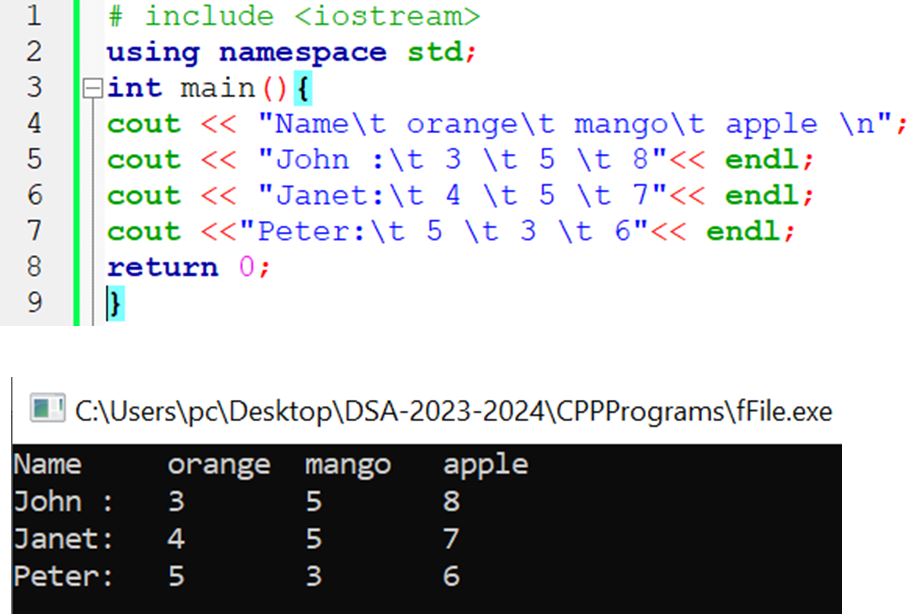


**e) Format Output using Escape Sequence**

Output formatting can also be accomplished using a combination of a backslash and a character, known as escape sequence.

| Escape | Meaning | Description |
| --- | --- | --- |
| \n | New line | Forces the cursor or insertion pointer to move to a new line |
| \t | tab | Moves tabs horizontally to create uniform white spaces between outputs. |
| \b | backspace | Move the character backwards without erasing anything. |
| \v | Vertical tab | Moves tabs vertically to create uniform white spaces between outputs. |
| \r | Carriage return | Moves the cursor to the first column of the net line. |
| \f | form feed | Moves the cursor to the start on the next page. |

The following is an example of a C++ program and the produced output



**I.11. Solve problems using C++ functions**

**I.11.1. Fundamentals of C++ Functions**

In C++, the smallest component having independent functionality is known as a function.

Every C++ program has at least one function called main ( ) through which other functions interact with each other directly or indirectly. This interaction is made possible through function calls and parameter passing discussed later in this unit.

**I.11.1.1. Features of C++ Function**

Like in other structured programming languages, the following are characteristics of C++ functions:

* A function is a complete sub-program in itself that may contain input, processing and output logic.
* A function is designed to perform a well defined task.
* A function can be compiled, tested and debugged separately without the intervention of other functions.
* A function has only one entry and one exit point.
* A function can interact with other functions using a mechanism known as function call and parameter passing.
* A function is designed in such a manner that it can be used with different programs or software systems.
* The calling function is suspended during the execution of the called function.This implies that there is only one function in execution at any given time.
* Control is always returned to the caller when the function execution terminates.

**I.11.1.2 Types of functions in C++**

Functions may be classified into two categories namely: Library or (built-in)functions and user-defined functions. Library functions are compiled and put in C++ library to simplify programming tasks while user-defined functions are the functions that we write to create a modular program.

**I.11.1.2.1 Library functions**

So far, we have been writing programs by first including (importing) functions from C++ Standard Library. C++ Standard Library provides a collection of predefined functions for common input and output manipulation, calculations, error checking and many other useful operations.

To use a library function, we first include its header file, then use a function that passes a list of arguments from the calling portion of the program.

***For example, to find the square root of a number, we use square root function sqrt() as follows:***

**root = sqrt(16);**

The function sqrt() evaluates the square root of 16 and returns 4 which is then assigned to the root. In this section, we demonstrate how to use mathematical, string and character manipulation functions.

**Math Functions**

The C++ Library provides a collection of Math functions used to perform mathematical and trigonometric computations.

For example, to raise 5 to power 3, we use the **pow()** function as follows:

**power = pow(5,3);**//returns 125

Frequently used functions require the inclusion or importing of **<cmath>** or **<math.h>** header file using #include directive.

| **Function** | **Description** | **Example** |
| --- | --- | --- |
| **ceil(x)** | rounds x to the smallest integer that is greater than or equal to x (rounds up the nearest integer). | ceil(9.2) is 10. ceil (-9.8) is -9.  X is a floating value |
| **cos(x)** | cosine of x (x in radians) | cos(0.0) is 1 |
| **exp(x)** | exponential function | exp(1.0) is 2.718282 |
| **fabs(x)** | absolute value of x | fabs(5.0) is 5.0. fabs(-8.76) is 8.76 |
| **floor(x)** | rounds x to the largest integer that is smaller than or equal to x (rounds downs the nearest integer). | floor (9.2) is 9. floor(-9.8) is -10  X is a floating value |
| **fmod(x,y)** | remainder of x/y as a floating point | fmod(2.6, 1.2) is 0.2 |
| **log(x)** | natural logarithm of x (base e) | log(2.718282) is 1.0 |
| **log10(x)** | logarithm of x (base 10) | log10(100.0) is 2.0 |
| **pow(x,y)** | x raised to power y (x,y) | pow(2.7) is 128 |
| **sin(x)** | trigonometric sine of x (x in radians) | sin(0.0) is 0 |
| **sqrt(x)** | square root of x (where x is a non negative value) | sqrt(9.0) is 3.0 |
| **tan(x)** | tangent of x (x in radians) | tan(0.0) is 0 |

**fmax() and fmin()**

**fmax()** and **fmin()** functions are defined in the **cmath** header file.

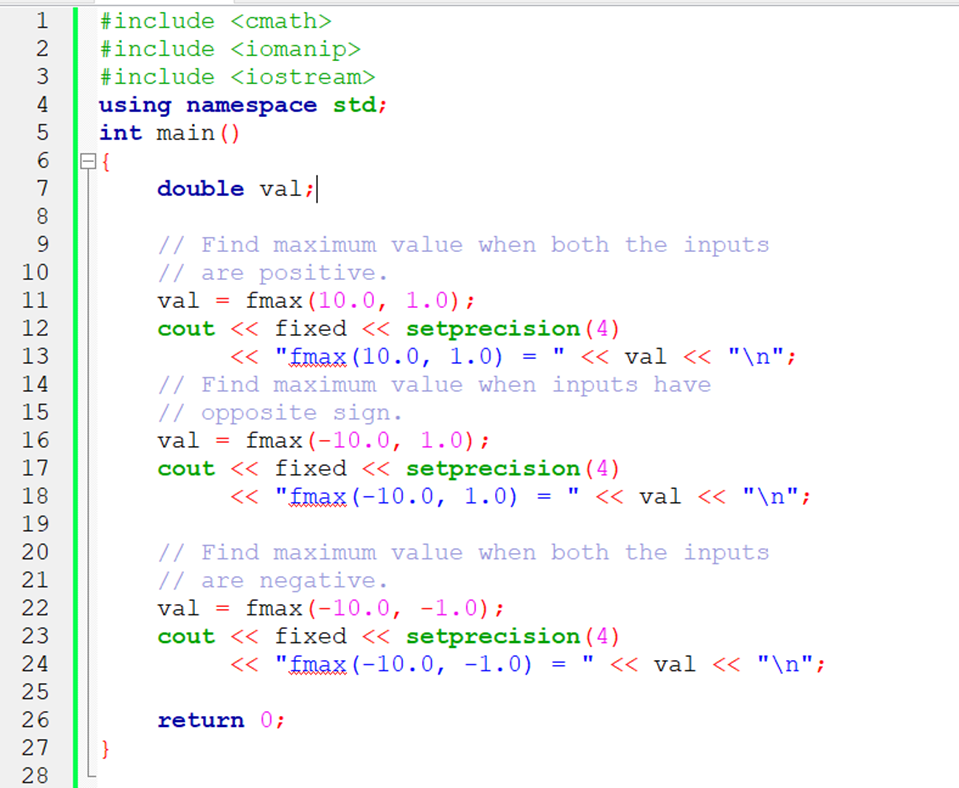
**fmax() function:** The syntax of this function is:

**double fmax (double x, double y);**

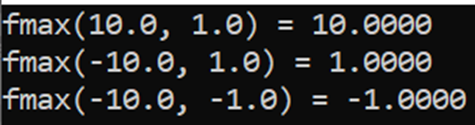
**float fmax (float x, float y);**

**long double fmax (long double x, long double y);**The input to this function are two values of type float, double or long double. The function returns the maximum of the two input values.

**Example:**



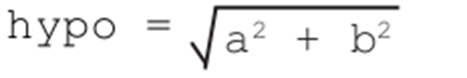
**Output**

****

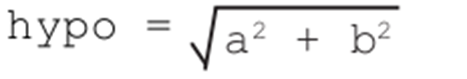
**fmin()** is the opposite of **fmax().** Both receive arguments in the same way, and **fmin()** returns the small argument.

**Exercises:**

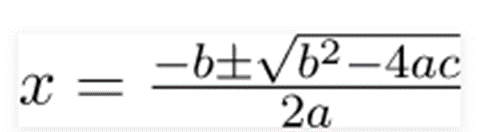
1. Using sqrt() and pow() calculate the hypotenuse of the right triangle.



1. Print all integer pairs(a,b) greater than 1 and less than 100 that can satisfy the hypotenuse rule. Hypotenuse should be also integer. Display the number of pairs found. Treat (3,4) and (4,3) as one pair.



1. Find the roots of quadratic function.



**Character functions**

Although a computer is a numerical machine, most often, data entered into a computer consists of numbers, characters and strings. The underlying fact is that characters are treated as integers. The C++ Library has in-built functions used to manipulate characters. The functions can be accessed by including **<cctype>** header file.

For example, to convert a character c from uppercase to lower case, we use the following statement:

*letter = tolower(c)*

**Activity:** Prompt the user to enter a character and get it converted to lowercase

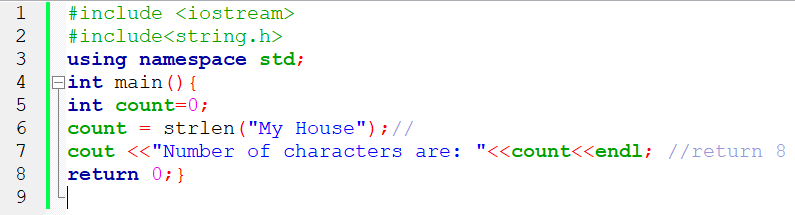
| **Function** | **Description** | **Example** |
| --- | --- | --- |
| isdigit(c) | Check whether c is a numeric digit | isdigit(‘5’)//returns 1 |
| isalpha(c) | Check whether c is a letter | isalpha(‘5’)//returns 0 |
| isupper(c) | Check whether c is in uppercase | isupper(‘x’)//returns 0 |
| tolower(c) | Converts c to lowercase | tolower(‘R’)//returns r |
| toupper(c) | Converts c to uppercase | toupper(‘r’)//returns R |
| isblank(c) | Returns true if character is a space or tab else returns false. | isblank(‘ ‘) or isblank(‘\t’) return 1 |
| isspace(c) | Returns true if character is a space or tab or whitespace control code ( Eg.\n,\r ) else returns false | isspace(‘\n’) return 1 |
| iscntrl(c) | Returns true if character is tab or any control codeelse returns false. | iscntrl(‘\t’) return 1 |

**String functions**

String is a collection of characters. There are two types of strings commonly used in C++ programming language:

* Strings that are objects of string class (The Standard C++ Library string class)
* Character array or C-strings (C-style Strings)

The string-handling library provides many useful functions for manipulating string data, comparing strings, searching strings for characters and substrings. To use the string manipulation functions, you must include the <cstring> or <string.h> header file. For example, the following statement returns the number of characters in “My House” string:



**String vs character array (c-string)**

A character array is simply an **array of characters** that can be terminated by a null character.

***Example:***

***char str[11]=”C++”;***

***char str[ ]=”C++”;***

***char \*str=”C++”*;**

**char str[] = {'C','+','+','\0'};**

**char str[4] = {'C','+','+','\0'};**

String is a class that characterizes objects that can be depicted as a stream of characters.

**Example:** String s=”I like c++”;

The sizeof character array needs to be designated statically. More memory can’t be distributed at run time whenever required.

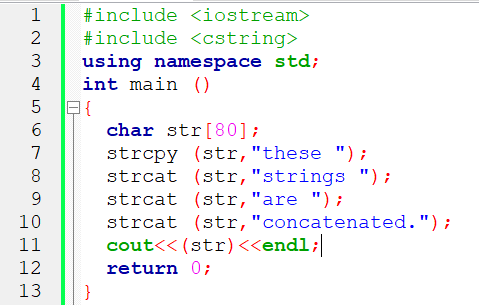
On string memory is distributed progressively and more memory can be distributed at run time.

Character array are quicker while strings are slower. Character array does not offer a lot inbuilt capabilities to control strings. String class has various functions that permit complex tasks on string.

**Example of commonly used string functions**

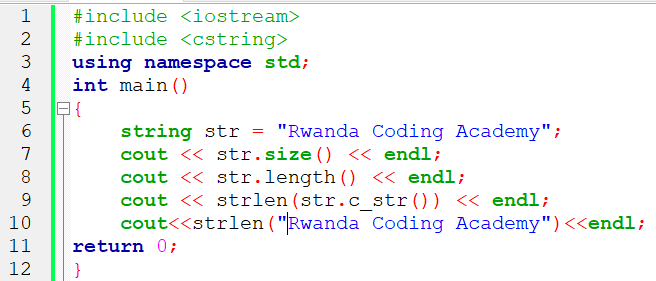
| **Function** | **Description** | **Example** |
| --- | --- | --- |
| **strcat(x,y)** | Concatenates two strings declared as an array of characters. The second string is appended to the first string.  Char str[]=”This is an”; char st[]=”example”; | strcat(str, st)  Return **This is an example** |
| **strcmp(x,y)** | Compare two strings  Return <0,=0 or >0 depending on character value. | strcomp(“he”,”se”) return -1  strcomp(“she”,”se”) return 1  strcomp(“se”,”se”) return 0 |
| **strlen (s)** | Returns the length of the string. strlen (s) does not work on string variables. It is a c function | strlen(‘him’)// returns 3 |
| **s.size()** | Return the length of the string. | String s=”him”; s.size()//return 3 |
| **s.length()** | Return the length of the string. | String s=”him”; s.length()//return 3 |
| **strcpy(x,y)** | Copies the second string to the first string. It overrides the value of the first string. | strcpy(y,x); copy x to y |
| **getline()** | This function is used to store a stream of characters as entered by the user in the object memory | getline(cin, x)// x as a stream of characters entered |
| **s.push\_back('c');** | Insert a character c from the end | String s=”hi”; s.push\_back(‘m’);// return him |
| **s.pop\_back()** | Delete a character at the end of a string | String s=”him”; s.pop\_back(s);// return hi |

**Examples:**



**Output:**





**Output:**

****

**Exercises:**

1. Using a while loop calculates the length of a string without using built-in functions.
2. Write a program that checks if a given string is a palindrome with and without built in functions.
3. Write a program that converts the entered string to Uppercase
4. Write a program that removes all spaces in a string and print the resulting string.

**I.11.1.2.2. User defined functions**

C++ allows programmers to define their own function(s). A user-defined function groups code to perform a specific task and that group of code is given a name(identifier). When the function is invoked from any part of the program, it all executes the codes defined in the body of function.

Creating user-defined functions requires that you declare the function name, return type, and list of arguments. After the declaration, you can then define the function as follows:

***return\_type fun\_name(parameters’ list and their types){***

***statements***

***}***

**Example:**

*int fun\_add(int a, int b){*

*int sum;*

*sum =a+b;*

*return sum;*

*}*

**Function declaration**

C++ requires that a function be defined before being called by the main () function

or any other function. For example, the addition function in our previous example comes before main. However, if you do not want to fully implement a function, you can first declare it and implement it later.

To declare a function without implementing the body, write the function return type, name and parameter list followed by a semicolon at the end of the statement.

The portion of a function that includes only the function name and list of arguments is called a function signature or prototype.

***Example:***

***double maximum(double x,double y,double z);***

**Function with no return type**

Previously, we defined the function to return an integer. But what if the function does not need to return a value? In this case, the type to be used is void, which is a special type to represent the absence of value. For example, a function that simply prints a message may not need to return any value:

***#include <iostream>***

***using namespace std;***

***void printMessage ()***

***{***

***cout << "I'm a function!";***

***}***

***int main ()***

***{***

***printMessage ();***

***}***

**Call a function**

When a function is called by another function, execution is transferred to the function until the return statement or end of function is encountered.

The function can be called any number of times.

**Overloading a function in C++**

Function overloading is a feature of object-oriented programming where two or more functions can have the same name but different parameters. When a function name is overloaded with different jobs it is called Function Overloading. In Function Overloading “Function” name should be the same and the arguments should be different. Function overloading can be considered as an example of a polymorphism feature in C++.

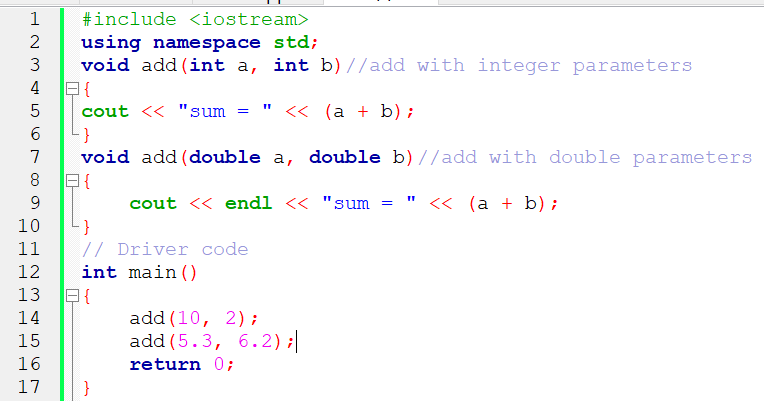
If multiple functions having same name but parameters of the functions should be different is known as Function Overloading.  
If we have to perform only one operation and having same name of the functions increases the readability of the program.  
Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the function such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you to understand the behavior of the function because its name differs.

The parameters should follow any one or more than one of the following conditions for Function overloading:

* **Parameters with different types**

***add(int a, int b)  
add(double a, double b)***

**Example:**

****

**Output:**

****

* **Different number of parameters**

***add(int a, int b)  
add(int a, int b, int c)***

* **Different sequence of parameters.**

***add(int a, double b)  
add(double a, int b)***

**Recursive function**

Recursion in computer science means a function is calling itself. We will use recursion whenever the solution of a problem depends on the solution of the small problem of the same nature.

Example: Factorial of n

n!=n\*n-1\*n-2\*n-3\*n-4\*n-5……..\*1

n!=n\*(n-1)!

fact(n)=n\*fact(n-1);

Big=small->smaller->smallest

**Recursion and Principle of Mathematical Induction(PMI)**

The Principle of Mathematical Induction (PMI) is a theorem that gives a method for establishing the truth of statements quantified over all integers greater than or equal to some given integer.

The Principle of Mathematical Induction (PMI) is just the following observation. Let P(n) be a statement for each positive integer n. If P(1) is true and if P(k) is true to prove that P(k +1) is true for all positive integers k, then P(n) is true for all positive integers n. In other words, if P(1) and P(k) ⇒ P(k + 1) then P(1) ⇒ P(2) ⇒ P(3) ⇒ P(4) ⇒ · · · .

It uses three main steps

1. **A Basis or Base case**
2. **Induction Hypothesis which is an assumption**
3. **Induction Step which use the induction hypothesis to argue for all value of n**

In computer science, PMI helps in the analysis of algorithms. But not only that,proofs by induction also tend to imply recursive algorithms for solving the problem at hand. **PMI is a main tool in proving the correctness of recursive algorithms.** The recursion works on the Principle of Mathematics Induction which is used to prove facts.

The recursion works on the Principle of Mathematics Induction which is used to prove facts. It has only 2 steps:

● ***Step 1. Show it is true for the first one***

● ***Step 2. Show that if any one is true then the next one is true***

Then all are true

In the world of numbers we say:

● Step 1. Show it is true for first case, usually **n=1**

● Step 2. Show that if **n=k** is true then **n=k+1** is also true

**PMI Example:Sum of Numbers from 1 to n**

Prove that ∑n=n(n+1)/2

1. Best case

Let us think n=0,

Left hand side is ∑0=0

Right Hand Side is n(n+1)/2=0(0+1)/2=0

When n=1, LHS is ∑1=1 , the RHS is 1(1+1)/2=2/2=1

1. Induction Hypothesis( Assumption) that ∑k=k(k+1) /2 is true
2. Induction step

∑(k+1)=(k+1)(k+2)/2

LHS =∑(k+1)

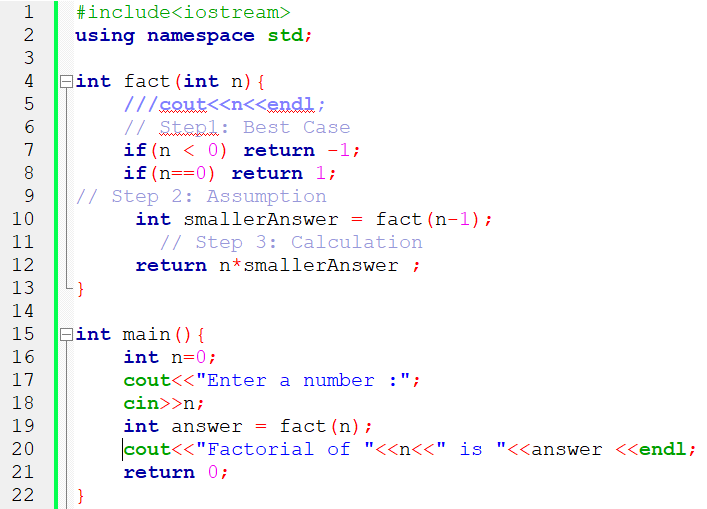
(k+1)+∑k then replace with the value of ∑k

(k+1)+k(k+1)/2=(k+1)\*2/2+k(k+1)/2

=(k+1)(k+2)/2

LHS=RHS Hence proved

**Factorial of a number:**

****

**Output**

****

Note: The two lines  are added to avoid the segmentation fault which occurs when memory is exhausted(Lack of memory) due to infinite calls.

**Fibonacci number at position n**

**Each number is equal to the sum of the preceding two numbers**. The Fibonacci sequence begins with the following 14 integers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, **55,** 89, 144, 233.

Ex: f(10)=f(9)+f(8)=55

f(n)=f(n-1)+f(n-2) which is a recursion as the big problem is solved with the solution of the small problems.

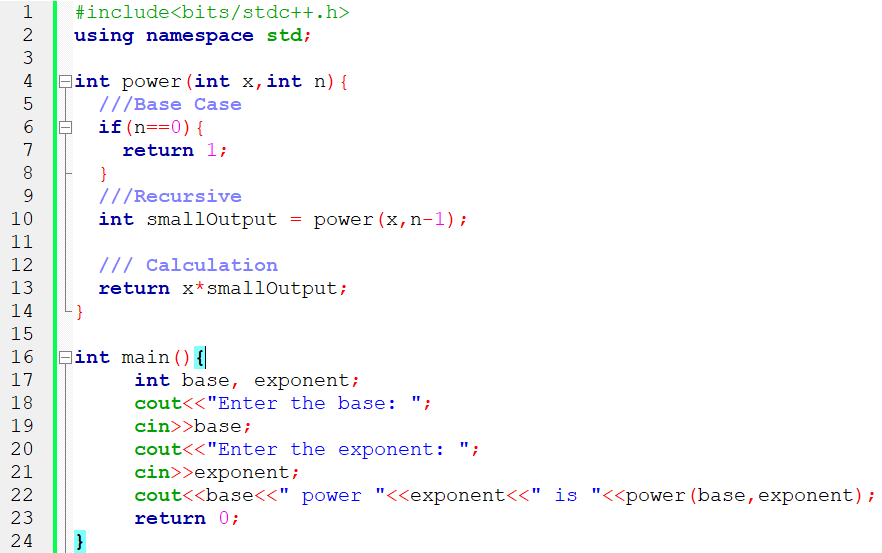
**Power of number**

To calculate the x to the power of n; we can use the recursive case, xn=x\*xn-1

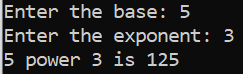
Base case

If n=0, x0=1

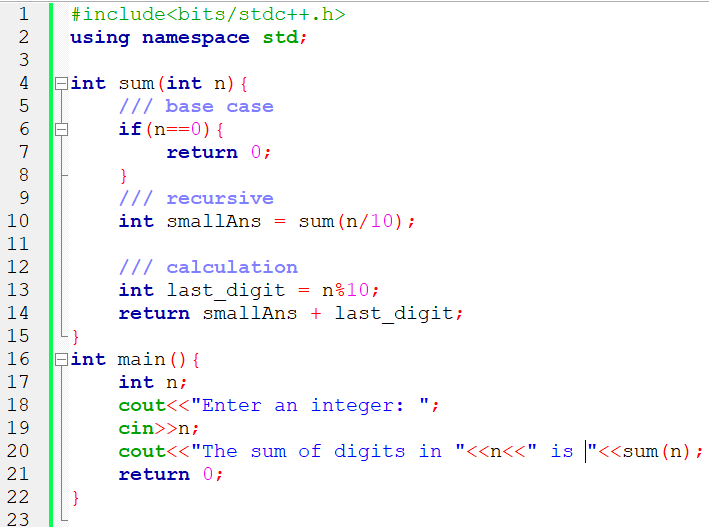
| **Calls** | (5,0) | (5,1) | (5,2) | (5,3) |
| --- | --- | --- | --- | --- |
| **Output** | 1 | 5 | 25 | 125 |



**Output:**



**Sum of digits on a number**



**Output**



**Print numbers from 1 to n**

Using recursive algorithms, write a function to print numbers from 1 to n.

If n is 3 the function will print

1 2 3

If n is 5 the function will print

1 2 3 4 5

If n is 10 the function will print

1 2 3 4 5 6 7 8 9 10

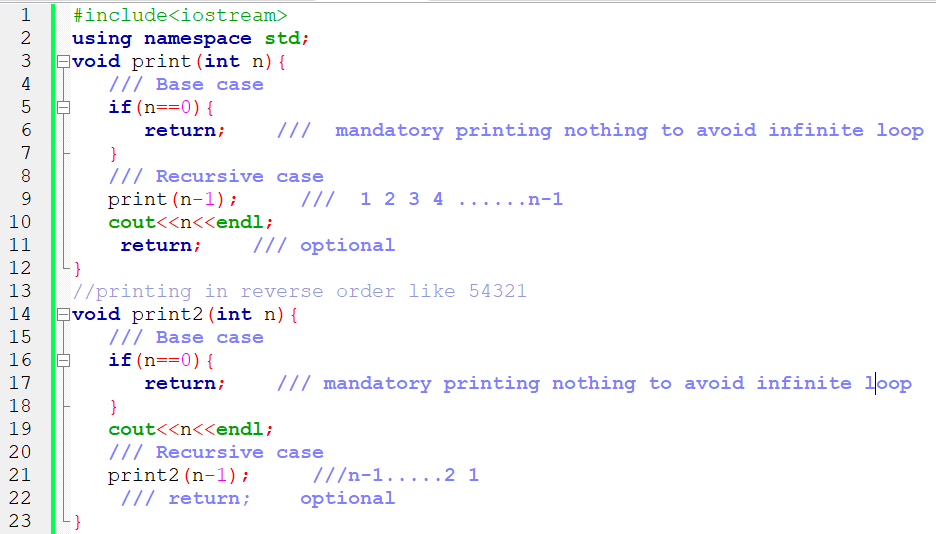
print(n)=print(n-1)+n;

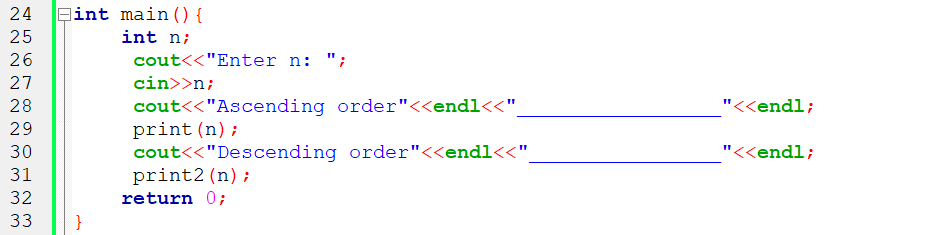
If the value of n is zero we will not print

If the value of n is 1 we will print 1

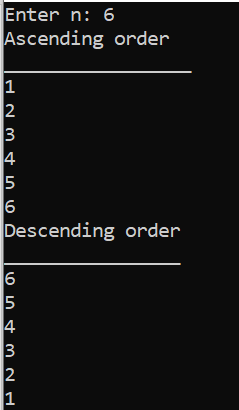
When printing in reverse order, the program will print.

5 4 3 2 1





**Output:**

****

**Count digits of a number:**

Build a function which counts the number of a given digit which is greater than or equal to 1.

If the number is 45756, the function returns 5

If the number is 900, the function returns 3

If the number is 15, the function returns 2

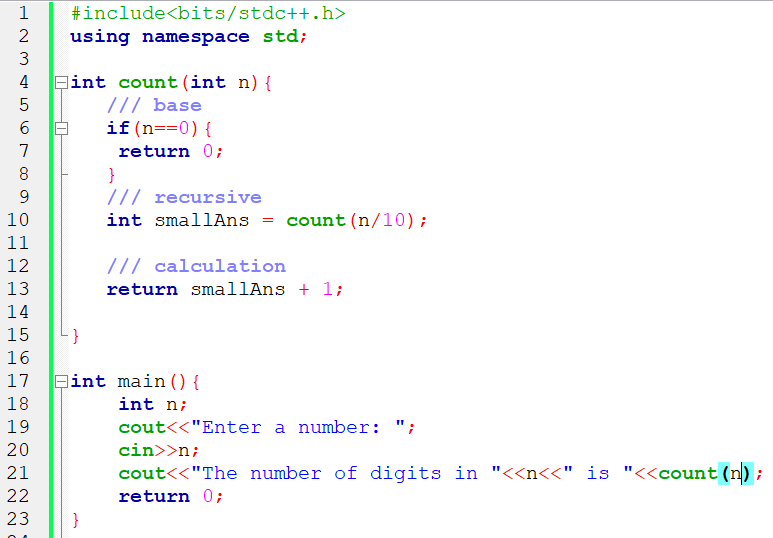
4125 will be broken into 412 and 5

Then 412 will be broken into 41 and 2

Then 41 will be broken into 4 and 1

Then 4 will be broken into 0 and 4, where 4 is our base case.

count(n)=count(n/10)+1, This would be our recursive case



**Output**

****

**Counting number of zeros in a given number:**

Build a function to calculate the number of zeros in a number n provided.

For example n=12050, countZeros(n)=2

Which means

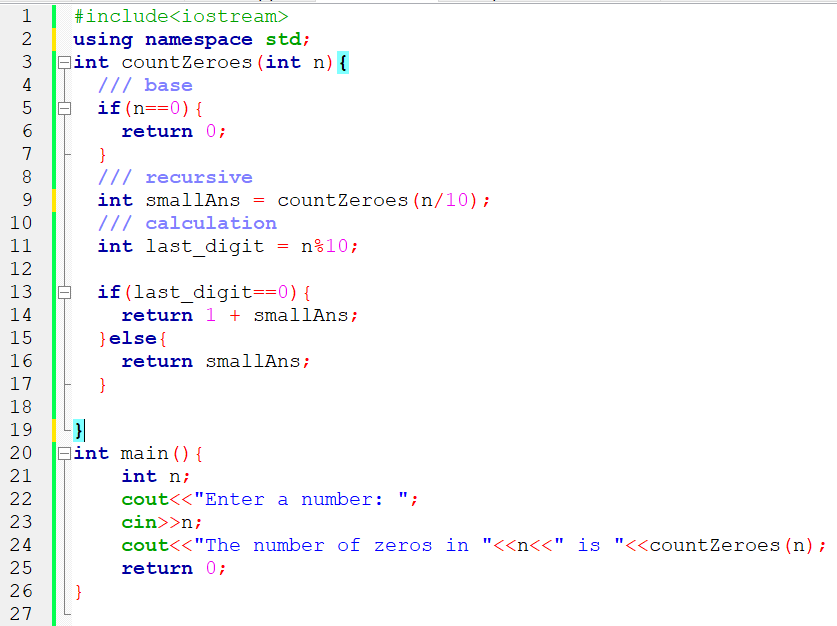
countZeros(n)=countZeros(n/10)

if(lastNumber =0)

Return smallAns+1;

Else

Return smallAns;



**Output:**

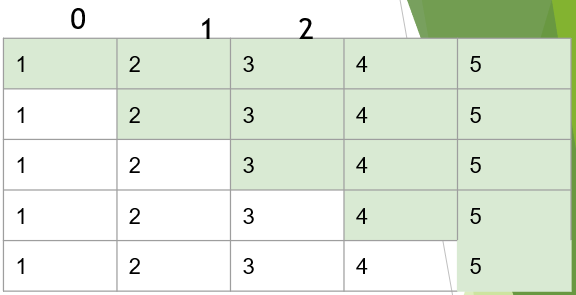


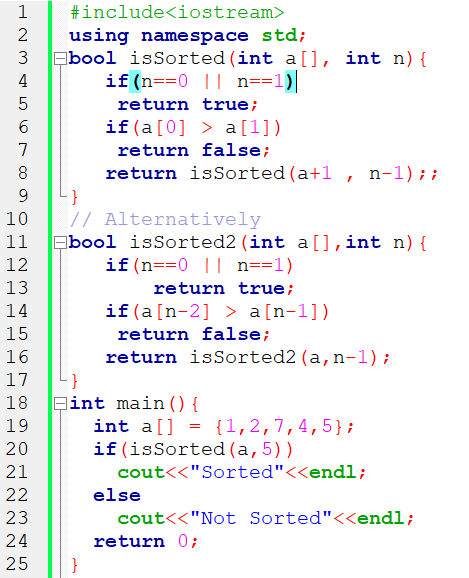
**Check if an array is sorted:**

Base case: if the number of elements is zero or 1, the array is sorted. Check if the first element is greater than the second , if so return false.

If true; use the recursion for a small array starting on the second index. The table below indicates the small arrays on each recursion by moving the pointer in our main array.

If any small array is not sorted, the big array is not sorted.





Output:



**Sum of array elements:**

We will return the sum of small array in green and add the first element



**Exercises:**

1. Write a program with a function that prints the character array.

Hint: if **abc** is provided, the program prints **abc.** Print the first character and call recursion on a small array. If the character is ‘\0’ return.

1. Write a program with a function that prints the reversed character array.

Hint: if **abc** is provided, the program prints **cba.** Print the last character and call recursion on a small array. If the character is ‘\0’ return.

1. Remove a character in the array.

Hint: if **abacada** is provided, by removing **a** the program print **bcd**

1. Replace the character in the array

Hint: if **abacada** is provided, by replacing **a** with **x** the program print **xbxcxdx**

1. Find the length of Character Array

Hint: if **abcde** i Finds provided, the length should be **5**

1. Convert the String of digits to Integer.

Hint: if “**12345**” is provided then the program would return **12345**

**Hint:**

If we have “12345” ,

Then call the recursion on “1234”,

Then call the recursion on “123”,

Then call the recursion on “12”,

Then call the recursion on “1”, now n=1, Then call the recursion on empty array.

1. if n=0, then return 0;
2. The small answer is 0, then 0\*10+1 as the last digit which is 1
3. Then 1\*10+2 as the last digit
4. Then 12\*10+3
5. Then 123\*10+4
6. Finally 1234\*10+5

8. A function that remove consecutive duplicate in the string

**Examples:**

if aabbbccdd the function would print abcd

if abcde is provided, the function would give abcde

9. Find the last index of an element in the array

a[6]={5,5,6,20,5,6}

if x=5, the last index of x is 4.

int lastIndex(int arr[], int element)

10. Find the first index of an element in the array

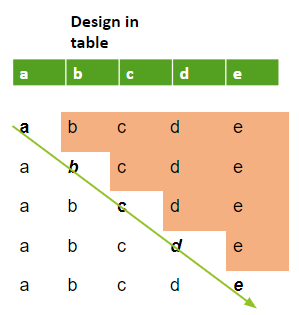
a[6]={5,5,6,20,5,6}

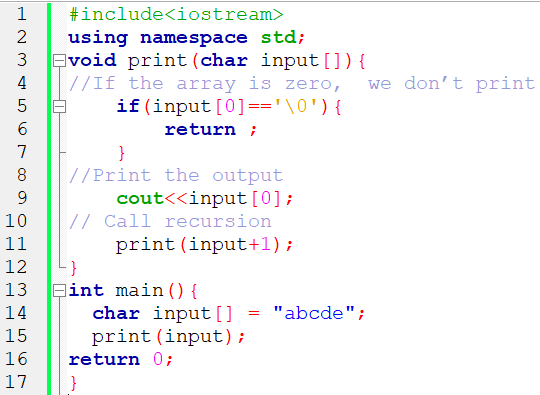
if x=5, the first index of x is 0.

int firstIdex(int arr[], int element)

**Recursion on Character Array: Print array and Print reversed array**

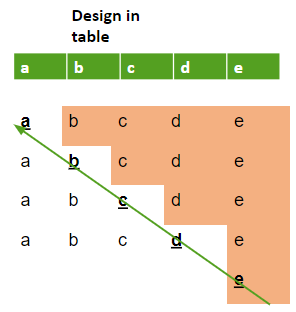
1. Let us think of two functions that (1) print a string/character array and (2) a reversed string . If we have the original string **abc** the reverse will be **cba.**
2. Let us start with first function
   1. The best case for the first function is when the array is empty. The first character will be equal to zero i.e arr[0]=‘\0’
   2. If the array is empty, we will not print anything.
   3. If the array is not empty, we will print the first element.
   4. Then call print recursion

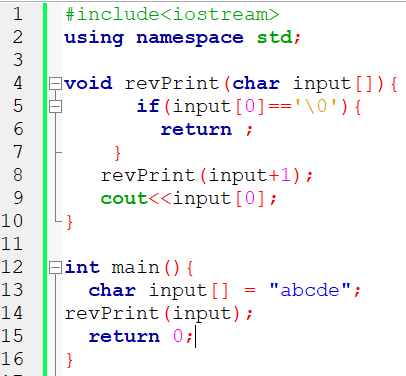




**Function 2: Recursion on Reversing Character Array**

* Let us start with second reverse function
* We first print the small array and the print the first value. The order will be reversed.

****



**Output**

****

**Recursion on Finding length of Character Array**

We can find the length of the character array using loops but we need to apply recursion.

If the character array is **abcde** the length should be **5**

**Array**

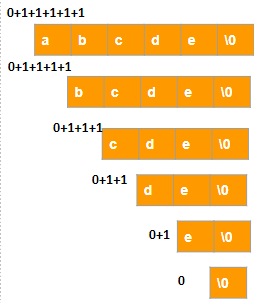


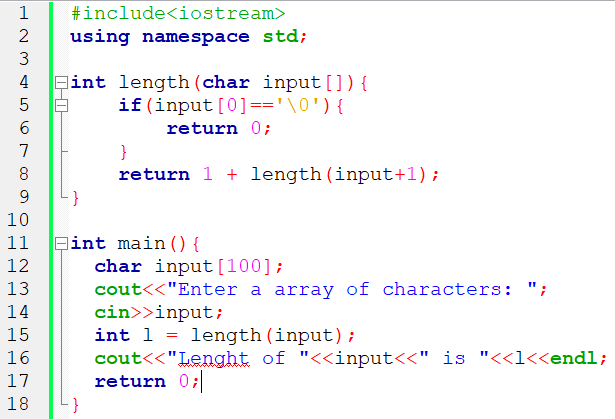
1+length

For the recursion, we need to call the length of the small array plus 1 of the first element at index 0.

The base case is when the array is empty, the function should return 0

If not, we calculate the value of length of the small array plus 1





**Output:**



**Replace a character recursively**

Basically, to replace the character **a** by **c** in ababa, means we need to modify the string to have **c**b**c**b**c**

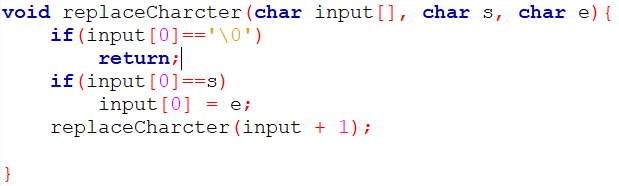
Simply, I need to :

* Check if the first character is **a**
* Replace the character a with **x**

**input[0]=’a’**

**input[0]=’c’;**

Then call the recursion for the small array by moving the pointer

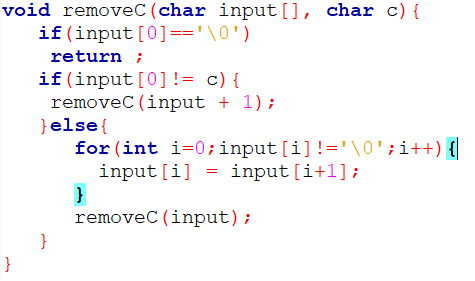


**Remove a character in array Recursively**

The next problem is to remove a character in the array. The array is **abacada**, then we want to remove the character **a** in the array to have **bcd**.

**abacada\0=> bcd\0**

if the first character is the target, we will shift the elements, otherwise we will do the recursion



**Convert a string digits to Integer**

Convert a string “12345” to 12345. This can be achieved using loops but the current target is the use of recursion with the following:

1. if n is 0, then return 0.
2. Calculate the small answer by calling a function on a small string by moving the last index.

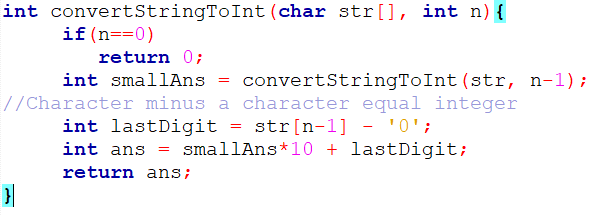
convertStrToInt(str,n-1);

Calculate the last digit

lastDigit=str[n-1]-’\0’

3. Multiply the smallAnswer by 10 and add the last digit to find the answer.

4. smallAnswer\*10+lastDigit



**Exercises on functions:** [**https://shorturl.at/ACDPW**](https://shorturl.at/ACDPW)

**I.12. Object Oriented Programming in C++**

**I.12.1 Introduction**

**OOP(Object Oriented Programming)** is a programming style that involves class and object. An object is a basic entity that possesses some properties and functions. In the real world, we are surrounded by Objects.

For example laptop is an object, chair is an object, table is an object.

We want to involve objects in our code because we want our code to be close to the real world.

**Object-oriented programming** aims to implement real-world entities like inheritance, hiding, polymorphism. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

**I.12.2 Class**

A class is a user-defined type which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is defined also as a blueprint or a **template** for **creating objects.**

It is composed of built in types, other user defined types and functions. The parts used to define class are called members. A class has zero or more members.

Members are **data members** and **member Functions**. Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions define the properties and behavior of the objects in a Class.

We access members using **object.member** notation. For example, if we defined a class X

X var; *//var is a variable of type X*

var.m=7; *//assign to var’s data member to m*

int x=var.mf(9);*// call var’s member function mf()*

You can read **var.m** as **var’s m.** Most people pronounce var dot m or var’s m. A member function **X’s mf()** does not need to use the var.m notation, It can use the plain member m as per previous definition int mf(int v){int d=**m**;..}

**I.12.2.1 Defining a class**

Classes are defined using either keyword class or keyword struct,

with the following syntax:

class class\_name {

access\_specifier\_1:

member1;

access\_specifier\_2:

member2;

...

} object\_names;

Where **class\_name** is a valid identifier for the class, **object\_names** is an optional list of names for objects of this class. The body of the declaration can contain ***members***, which can either be **data** or **function** declarations, and optionally *access specifiers*.

Classes have the same format as **plain *data structures***, except that they can also include functions and have ***access specifiers*.** An ***access specifier*** is one of the following three keywords: **private, public** or **protected**. These specifiers modify the access rights for the members that follow them:

* **private** members of a class are accessible only from within other members of the same class (or from their *"friends"*).
* **protected** members are accessible from other members of the same class (or from their *"friends"*), but also from members of their derived classes.
* Finally, **public** members are accessible from anywhere where the object is visible.

By default, all members of a class declared with the class keyword have ***private*** access for all its members. Therefore, any member that is declared before any other *access specifier* has private access automatically. For example:

***class Rectangle {***

***int width, height;***

***public:***

***void set\_values (int,int);***

***int area (void);***

***} rect;***

Declares a class (i.e., a type) called **Rectangle** and an object (i.e., a variable) of this class, called **rect**. This class contains **four** **members**: two data members of type int (member width and member height) with ***private*** *access* (because private is the default access level) and two member functions with *public access*: the functions set\_values and area, of which for now we have only included their declaration, but not their definition.

Notice the difference between the *class name* and the *object name*: In the previous example, Rectangle was the *class name* (i.e., the type), whereas rect was an object of type Rectangle. It is the same relationship int and a have in the following declaration:

**int a;**

After the declarations of Rectangle and **rect**, any of the public members of object **rect** can be accessed as if they were normal functions or normal variables, by simply inserting a dot (.) between ***object name*** and ***member name*.** This follows the same syntax as accessing the members of plain data structures. For example:

rect.set\_values (3,4);

myarea = rect.area();

The only members of rect that cannot be accessed from outside the class are **width** and **height**, since they have private access and they can only be referred to from within other members of that same class.

**I.12.3 Object**

An Object is an identifiable entity with some characteristics and behaviour. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated. Classes are an expanded concept of data structures: like data structures, they can contain data members, but they can also contain functions as members. An object is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.

***Examples:***

*class Rectangle {*

*int width, height;*

*public:*

*void set\_values (int,int);*

*int area (void);*

*} rect; // rect is an object*

*class Person*

*{*

*char name[20];*

*int id;*

*public:*

*void getdetails(){}*

*};*

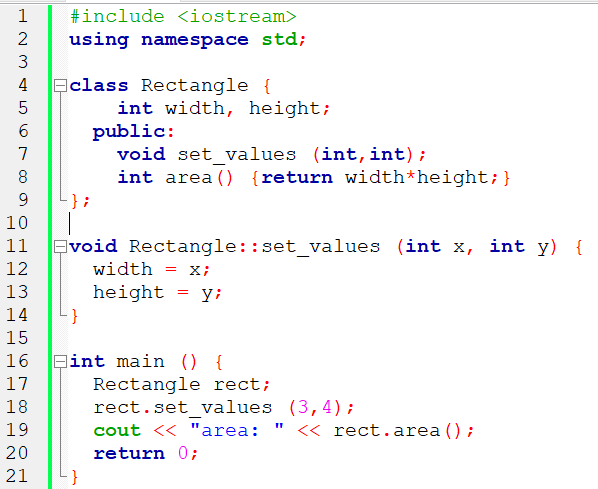
*int main()*

*{*

*person p1; // p1 is a object*

*}*

**Implementation of Rectangle class**



**Output:**

****

This example reintroduces the *scope operator* (**::**, two colons), seen in earlier chapters in relation to namespaces. Here it is used in the definition of function **set\_values** to define a member of a class outside the class itself.

Notice that the definition of the member function **area** has been included directly within the definition of class **Rectangle** given its extreme simplicity. Conversely, **set\_values** it is merely declared with its prototype within the class, but its definition is outside it. In this outside definition, the operator of scope (::) is used to specify that the function being defined is a member of the class Rectangle and not a regular non-member function.

The scope operator (::) specifies the class to which the member being defined belongs, granting exactly the same scope properties as if this function definition was directly included within the class definition. For example, the function **set\_values** in the previous example has access to the variables width and height, which are private members of class Rectangle, and thus only accessible from other members of the class, such as this.

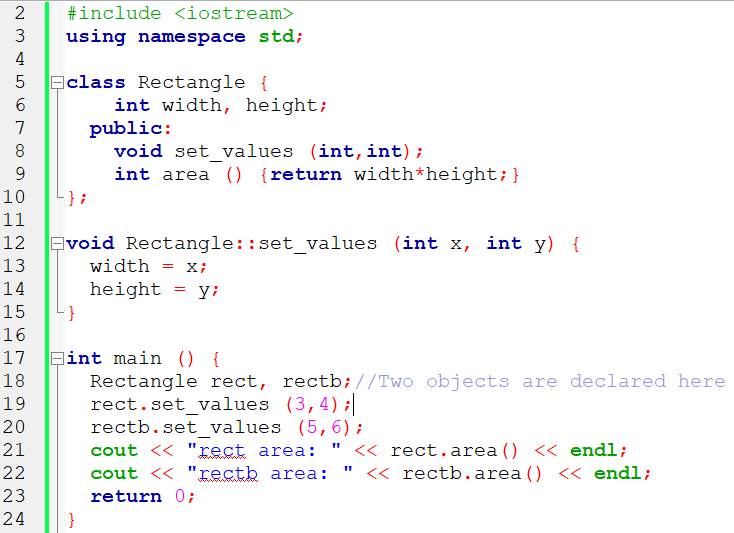
The only difference between ***defining a member function completely within the class definition or to just include its declaration in the function and define it later outside the class***, is that in the first case the function is automatically considered an *inline* member function by the compiler, while in the second it is a normal (not-inline) class member function. This causes no differences in behavior, but only on possible compiler optimizations.

Members **width** and **height** have private access (remember that if nothing else is specified, all members of a class defined with keyword class have **private** access). By declaring them private, access from outside the class is not allowed.

This makes sense, since we have already defined a member function to set values for those members within the object: the member function **set\_values**. Therefore, the rest of the program does not need to have direct access to them.

**Multiple objects**

The most important property of a class is that it is a type, and as such, we can declare multiple objects of it. For example, following with the previous example of class **Rectangle**, we could have declared the object **rectb** in addition to object **rect**:



**Output**



The class (type of the objects) is **Rectangle**, of which there are two instances (i.e., objects): **rect** and **rectb**. Each one of them has its own member variables and member functions.

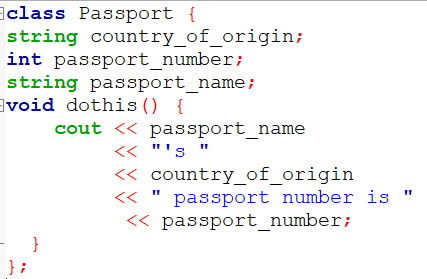
Notice that the call to **rect.area()** does not give the same result as the call to **rectb.area().** This is because **each object of class Rectangle has its own variables width and height**, as they -in some way- have also their own function members **set\_value** and **area** that operate on the object's own member variables.

Classes allow programming using object-oriented paradigms: **Data and functions are both members** of the object, reducing the need to pass and carry handlers or other state variables as arguments to functions, because they are part of the object whose member is called. Notice that **no arguments were passed on the calls to rect.area or rectb.area**. Those member functions directly used the data members of their respective objects **rect** and **rectb**.

**I.12.4 Encapsulation**

In C++, encapsulation involves combining similar data and functions into a single unit called a class. By encapsulating these functions and data, we protect that data from change. This concept is also known as data or information hiding.

Let’s look at an example of C++ class encapsulation:



Within the class Passport, we declared the variables and methods that we wanted to include. Note that we can only use cout() if we wrap it in a function, as classes can execute [member functions](https://www.udacity.com/blog/2021/07/cpp-class-methods-explained.html) but not objects.

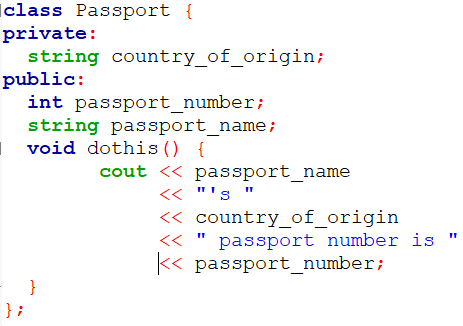
**Advantages of encapsulation**

Keeping pertinent code encapsulated within a class helps streamline your program. You can contain your code in one location so your code is easier to follow.

This, in turn, makes your program easier to debug. If a function within your encapsulated class is not working correctly, you’ll likely be able to pinpoint the problem to the class instead of having to comb through your code.

When you encapsulate, you can control the flow of data into and out of a class. Classes allow you to make certain functions or variables inaccessible to protect them from being used or changed outside of the class. You can also make specific components accessible to use elsewhere in your code.

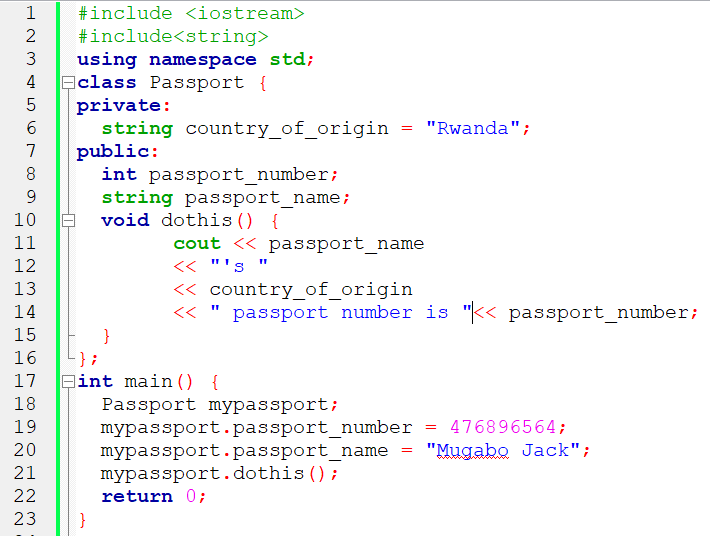
Let’s look at the tools that allow you to protect or enable access to a class’s variables or functions.



**Printing From an Encapsulated Class in C++**

The above example is a valid class, but it has some limitations. We are free to use or assign values to **passport\_number** and **passport\_name** in an instance of the **Passport** class anywhere else in our code. This also leaves these variables exposed, allowing other parts of the codebase to manipulate them in a way we had not intended.

However, the **country\_of\_origin**string is private and therefore inaccessible by the rest of our program. We can assign **country\_of\_origin**a value as we declare it, but this means we cannot change it elsewhere:



**Output**

****

**Access encapsulated data in C++**

C++ allows access to private data members and functions inside a class through the use of getters and setters. get()and set()are methods, initialized in the public section of a class, as a way to either read or write to private data.

set() enables us to write a value to a private variable within our class. get(), on the other hand, allows a function outside of the class to read private data. Using set() and get() allows us to protect data members or functions in our class while still being able to use them.

Ideally, we want to make our passport information private to prevent other users from tampering with it. With setters and getters in C++, we can still modify that data.

We create a function called setpassport\_number() with one parameter that permits us to write a value to passport\_number, a private variable. Similarly, we introduce getpassport\_number() to read that value outside of our class.

Our completed class has three private variables and all the setters and getters we need to read and write to those variables. Note that setters and getters need to be public so that we can use them outside of our class. OurPassport class looks like this:

***#include <iostream>***

***using namespace std;***

***class Passport {***

***int passport\_number;***

***string passport\_name;***

***string country\_of\_origin;***

***public:***

***void setPassport\_number(int passnum) {***

***passport\_number = passnum;***

***}***

***void setPassport\_name(string passname) {***

***passport\_name = passname;***

***}***

***void setCountry\_of\_origin(string origin) {***

***country\_of\_origin = origin;***

***}***

***int getPassport\_number() {***

***return passport\_number;***

***}***

***string getPassport\_name() {***

***return passport\_name;***

***}***

***string getCountry\_of\_origin() {***

***return country\_of\_origin;***

***}***

***};***

***int main() {***

***Passport mypassport;***

***mypassport.setPassport\_number(476896564);***

***mypassport.setPassport\_name("Mugabo Jack");***

***mypassport.setCountry\_of\_origin("Rwanda");***

***cout << mypassport.getPassport\_name() << "'s " << mypassport.getCountry\_of\_origin()***

***<< " passport number is " << mypassport.getPassport\_number();***

***return 0;***

***}***

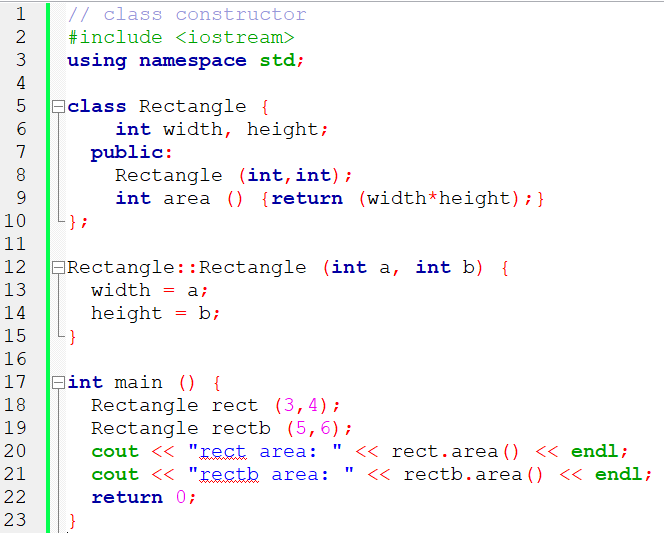
**I.12.5 Constructors**

What would happen in the previous example if we called the member function **area** before having called **set\_values**? An undetermined result, since the members **width** and **height** had never been assigned a value.

In order to avoid that, a class can include a special function called its ***constructor***, which is automatically called whenever a new object of this class is created, allowing the class to initialize member variables or allocate storage.

This constructor function is declared just like a regular member function, but with a **name** that matches the **class name** and **without any return type; not even void.**

The **Rectangle** class above can easily be improved by implementing a constructor:



The results of this example are identical to those of the previous example. But now, class Rectangle has no member function set\_values, and instead has a constructor that performs a similar action: it initializes the values of width and height with the arguments passed to it.

Notice how these arguments are passed to the constructor at the moment at which the objects of this class are created:

Rectangle rect (3,4);

Rectangle rectb (5,6);

Constructors **cannot be called explicitly as if they were regular member functions.** They are only executed once, when a new object of that class is created.

Notice how neither the constructor prototype declaration (within the class) nor the latter constructor definition, **have no return type, even void**: Constructors never return values, they simply initialize the object.

In above paragraphs, three characteristics of constructors which make them special functions are mentioned:

1. Have the same name as a Class
2. No return type even void
3. Invoked directly when an object of class is created

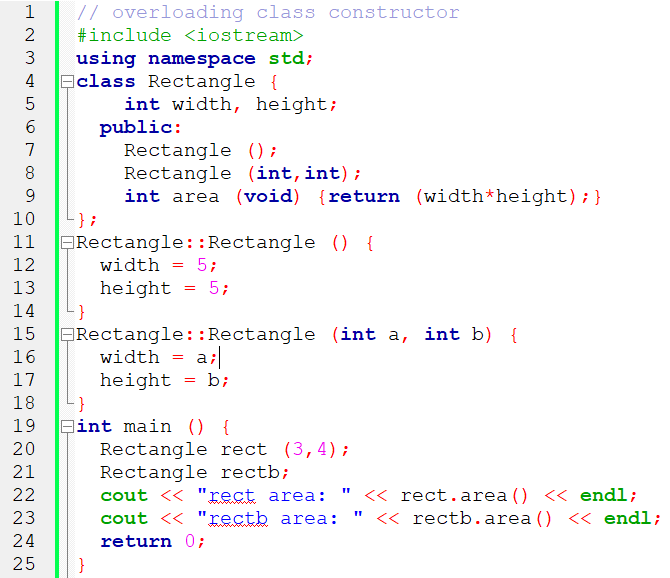
**Constructor overloading**

Like any other function, a constructor can also be overloaded with different versions taking different parameters: with a different number of parameters and/or parameters of different types. The compiler will automatically call the one whose parameters match the arguments.

On the left , two objects of class Rectangle are constructed: **rect** and **rectb**. rect is constructed with two arguments, like in the example before.

But this example also introduces a special kind **constructor**: the *default* ***constructor***. The *default constructor* is the constructor that takes no parameters, and it is special because it is called when an object is declared but is not initialized with any arguments.

In our current example the *default constructor* is called for **rectb**. Note how **rectb** is not even constructed with an empty set of parentheses - in fact, empty parentheses cannot be used to call the default constructor:



**Output:**

****

***Rectangle rectb; // default constructor called***

***Rectangle rectc(); // oops, default constructor NOT called***

This is because the empty set of parentheses would make of rectc a function declaration instead of an object declaration: It would be a function that takes no arguments and returns a value of type Rectangle.

**Uniform initialization**

The way of calling constructors by enclosing their arguments in parentheses, as shown above, is known as *functional form*. But constructors can also be called with other syntaxes:

First, constructors with a single parameter can be called using the variable initialization syntax (an equal sign followed by the argument):

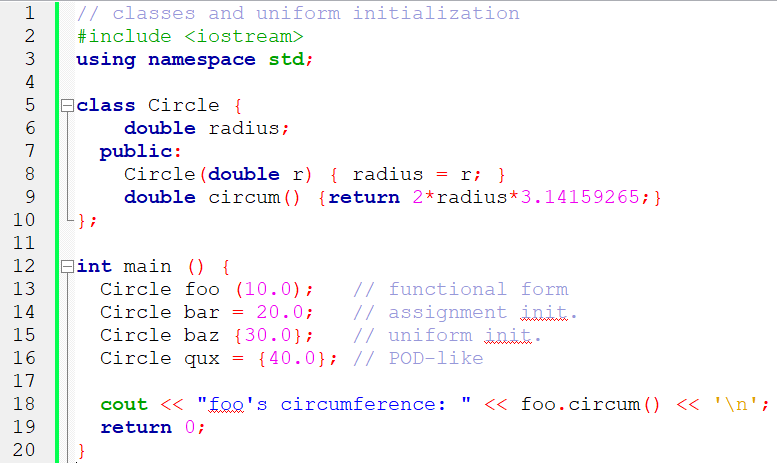
class\_name object\_name = initialization\_value;

More recently, C++ introduced the possibility of constructors to be called using *uniform initialization*, which essentially is the same as the functional form, but using braces ({}) instead of parentheses (()):

class\_name object\_name { value, value, value, ... }

Optionally, this last syntax can include an equal sign before the braces.

Here is an example with four ways to construct objects of a class whose constructor takes a single parameter:

****

**Output**

****

An advantage of uniform initialization over functional form is that, unlike parentheses, braces cannot be confused with function declarations, and thus can be used to explicitly call default constructors:

***Rectangle rectb; // default constructor called***

***Rectangle rectc(); // function declaration (default constructor NOT called)***

***Rectangle rectd{}; // default constructor called***

The choice of syntax to call constructors is largely a matter of style. Most existing code currently uses functional form, and some newer style guides suggest to choose uniform initialization over the others.

**Member initialization in constructors**

When a constructor is used to initialize other members, these other members can be initialized directly, without resorting to statements in its body. This is done by inserting, before the constructor's body, a colon (:) and a list of initializations for class members. For example, consider a class with the following declaration:

*class Rectangle {*

*int width,height;*

*public:*

*Rectangle(int,int);*

*int area() {return width\*height;}*

*};*

The constructor for this class could be defined, as usual, as

*Rectangle::Rectangle (int x, int y) { width=x; height=y; }*

**or**

*Rectangle::Rectangle (int x, int y) : width(x), height(y) { }*

Or

*Rectangle::Rectangle (int x, int y) : width(x) { height=y; }*

Note how in this last case, the constructor does nothing else than initialize its members, hence it has an empty function body.

For members of fundamental types, it makes no difference which of the ways above the constructor is defined, because they are not initialized by default, but for member objects (those whose type is a class), if they are not initialized after the colon, they are default-constructed.

Default-constructing all members of a class may or may always not be convenient: in some cases, this is a waste (when the member is then reinitialized otherwise in the constructor), but in some other cases, default-construction is not even possible (when the class does not have a default constructor). In these cases, members shall be initialized in the member initialization list. For example:

*// member initialization*

*#include <iostream>*

*using namespace std;*

*class Circle {*

*double radius;*

*public:*

*Circle(double r) : radius(r) { }*

*double area() {return radius\*radius\*3.14159265;}*

*};*

*class Cylinder {*

*Circle base;*

*double height;*

*public:*

*Cylinder(double r, double h) : base (r), height(h) {}*

*double volume() {return base.area() \* height;}*

*};*

*int main () {*

*Cylinder foo (10,20);*

*cout << "foo's volume: " << foo.volume() << '\n';*

*return 0;*

*}*

**I.12.6 Destructor**

Destructor is an instance member function that is invoked automatically whenever an object is going to be destroyed. Meaning, a destructor is the last function that is going to be called before an object is destroyed.

* A destructor is also a special member function like a constructor. Destructor destroys the class objects created by the constructor.
* Destructor has the same name as their class name preceded by a tilde (~) symbol.
* It is not possible to define more than one destructor.
* The destructor is only one way to destroy the object created by the constructor. Hence destructor can-not be overloaded.
* Destructor neither requires any argument nor returns any value.
* It is automatically called when an object goes out of scope.
* Destructor release memory space occupied by the objects created by the constructor.
* In a destructor, objects are destroyed in the reverse of an object creation.

The syntax for defining the destructor within the class:

*~ <class-name>() {*

*// some instructions*

*}*

The syntax for defining the destructor outside the class:

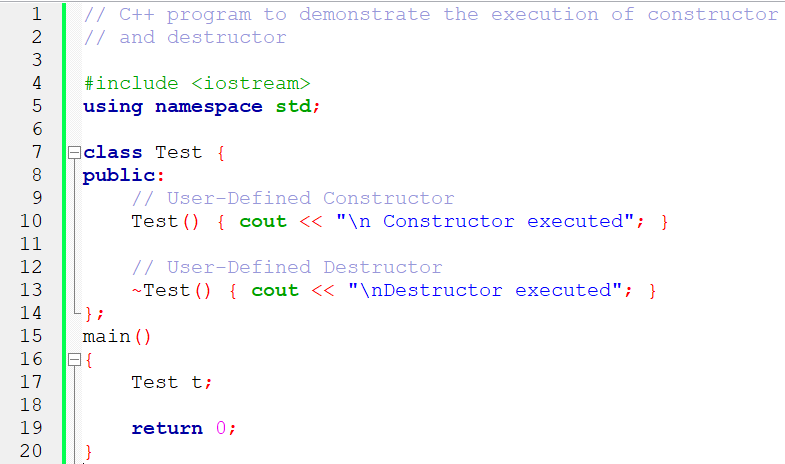
*<class-name> :: ~<class-name>() {*

*// some instructions*

*}*

**Example 1**

The below code demonstrates the automatic execution of constructors and destructors when objects are created and destroyed, respectively.



**I.12.7 Pointer to class**

Objects can also be pointed to by pointers: Once declared, a class becomes a valid type, so it can be used as the type pointed to by a pointer. For example:

Rectangle \* prect; **//is a pointer to an object of class Rectangle.**

Similarly as with plain data structures, the members of an object can be accessed directly from a pointer by using the arrow operator (->). Here is an example with some possible combinations:

// pointer to classes example

#include <iostream>

using namespace std;

class Rectangle {

int width, height;

public:

Rectangle(int x, int y) : width(x), height(y) {}

int area(void) { return width \* height; }

};

int main() {

Rectangle obj (3, 4);

Rectangle \* foo, \* bar, \* baz;

foo = &obj;

bar = new Rectangle (5, 6);

baz = new Rectangle[2] { {2,5}, {3,6} };

cout << "obj's area: " << obj.area() << '\n';

cout << "\*foo's area: " << foo->area() << '\n';

cout << "\*bar's area: " << bar->area() << '\n';

cout << "baz[0]'s area:" << baz[0].area() << '\n';

cout << "baz[1]'s area:" << baz[1].area() << '\n';

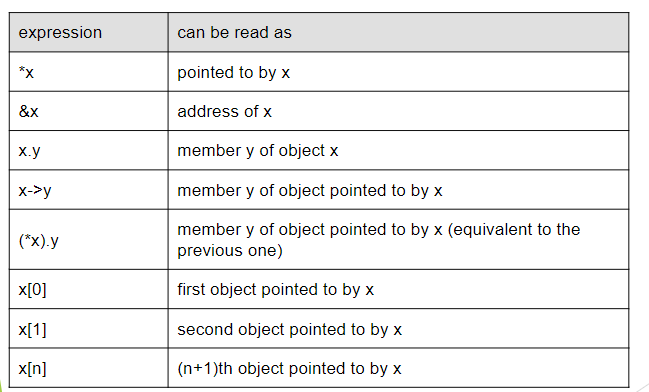
delete bar;

delete[] baz;

return 0;

}

**Operators to pointer and Object**



**I.12.8 Inheritances**

1. **ANALYZE DATA STRUCTURES AND ALGORITHMS**

**2.1. Algorithm Analysis**

**2.1.1 Introduction: What is an Algorithm?**

An **Algorithm** is a finite set of unambiguous steps or instructions to solve a given problem.

An algorithm is a **set of instructions for solving a problem** or accomplishing a task.

It can also be defined as a process or **set of rules to be followed** in calculations or other problem-solving operations, especially by a computer. Knowledge of algorithms helps us to get desired results faster by applying the appropriate algorithm. We learn by experience. With experience, it becomes easy to solve new problems.

**The properties of an algorithm are:**

1. It takes **zero or more inputs.**

2. It should produce **one or more output.**

3. It should be **Deterministic.** It produces the same output if the same input is provided again.

4. It should be **Correct.** It should be correct and able to process all the given inputs and provide the correct output.

5.It should **Terminate** in a finite time.

6. It should be **Efficient**. The algorithm should be efficient in solving problems.

**2.1.2 Algorithm representation**

An algorithm is presented by:

**Natural Language**

Using natural language with some predefined structures to describe the algorithm step by step. This method is useful when the target audience is not familiar with programming concepts.

**Pseudocode**

Using a simplified programming language to describe the steps of the algorithm. Pseudocode abstracts the details of a specific programming language and focuses on the logic of the algorithm.

**Flowchart**

Visual representations of the algorithm using different shapes to represent different steps, with arrows showing the flow of the process. This method is particularly useful for presenting complex algorithms in a more digestible format.

**2.1.3 Algorithm Complexity**

Algorithmic complexity is a very important concept that allows us to **compare algorithms**  in order to find which one is the most efficient. There is a standard notation called **big O** which allows you to measure the performance of an algorithm.

There are **two types of Complexity:**

1. First is **Time-Complexity**, how much time is required by an algorithm to produce output for an input of size 'n'.

Time-Complexity is represented by function T(n) - time required versus the input size n.

2. Second is **Space-Complexity**, how much RAM or memory that an algorithm is going to

consume to produce output for an input of size 'n'.

Space-Complexity is represented by function S(n) - memory used versus the input size n.

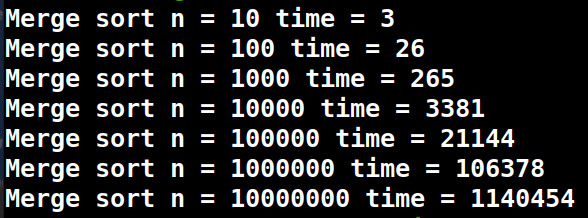
An algorithm is analyzed using Time Complexity and Space Complexity. Writing an efficient algorithm helps to consume the minimum amount of time for processing the logic.

**Experimental Analysis**

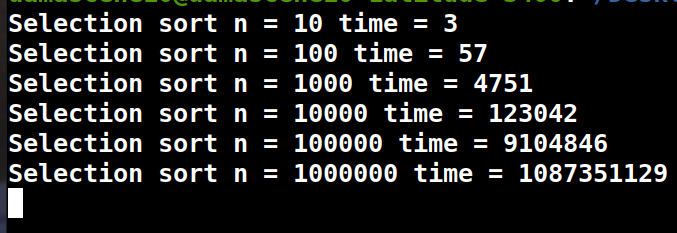
Write code for every solution, record the time and plot the time on the graph. Compile and run the following program for both Merge Sort and selection sort for 10 million elements: (find the code at [***https://shorturl.at/hsCJT***](https://shorturl.at/hsCJT)

)

**Merge Sort with 10.000.000 elements**



**Selection Sort with 10.000.000 elements**

****

**Observation:** The selection sort to finish sorting 100 Million elements will take 8-10 hours which is very difficult to wait or handle. Experimental analysis is difficult to use.

**Theoretical analysis of an Algorithm and categories of complexity**

**Exponential complexity:**

**Problem:** Suppose you forget the PIN made of three digits for your briefcase.

Rather than testing them out of order, because it would be impossible to remember all the attempts, we think about **different strategies** . One of them is to **try the smallest number (000) then the next number (001)** and so on until reaching 999.

Now let's see how to describe this problem with pseudo-code:

***START***

***VAR i, j, k, PIN\_to\_search\_for, combination***

***FOR i←0 TO 9***

***FOR j←0 TO 9***

***FOR k←0 TO 9***

***combination←CONCATENATE(i, j, k)***

***IF PIN\_to\_search\_for=combination THEN***

***BREAK***

***ENDIF***

***ENDFOR***

***ENDFOR***

***ENDFOR***

***STOP***

In this algorithm we have 3 loops going from 0 to 9, therefore 10 x 10 x 10 iterations, or 103 iterations . With Big O notation , we will say that the time complexity of this algorithm is ***O(103)***.

If the PIN has 3 digits , you must test 10\*10\*10=1,000 combinations (combinations between 000 and 999). On the other hand, if it’s made of 4 digits, you have to test 10\*10\*10\*10=10,000 combinations. If it had been 5 digits, we would have had to test 10\*10\*10\*10\*10=100,000 , and so on. We therefore see that our safe opening algorithm depends on the **number of digits** in the code, more precisely, the time taken to open the briefcase is multiplied by 10 each time we add a digit: 10 for a 1-digit code, 100=(10 x 10) for a 2-digit code, 1,000=(10 x 10 x 10) for a three-digit code, and so on.

If we note n the number of digits of our briefcase’s PIN, the calculation time is therefore **10n**; thus, using the notion **big O** notation, we can note that the time complexity of our algorithm is ***O(10n)***

We then say that **the complexity is exponential**.

**Linear complexity**

Suppose there is a click when you reach the correct digit while turning the dial for each digit to make the PIN. The technique is simple: **to turn the dial for the first number until hearing a “ *click* ”** . We then know that the number is good and move on to the next one. We can therefore find the correct numbers one by one, without having to worry about the others.

You realize that the time taken in this case is less than the one taken before.

When the PIN is made of 3 digits, we will therefore only have to test **30 combinations** (10 + 10 + 10, or 10 x 3) for this three-digit padlock (which is better than the 1000 combinations we were about to try before).

Now let's see how to describe this new solution with pseudo-code:

***START***

***VAR i, j, k, digit\_1, digit\_2, digit\_3 : INTEGERS,***

***listen\_click←TRUE: BOOL***

***FOR i←0 TO 9***

***IF listen\_click THEN***

***Digit\_1←i***

***ENDIF***

***ENDFOR***

***FOR j←0 TO 9***

***IF listen\_click THEN***

***Digit\_2←j***

***ENDIF***

***ENDFOR***

***FOR k←0 TO 9***

***IF listen\_click THEN***

***Digit\_3←k***

***ENDIF***

***ENDFOR***

***STOP***

If you look closely, we are testing **10 new combinations for each new number added to the number of digits** . In other words, if n is the number of digits, we test **10\*n** combinations. So with the **big O** notation , we can say that this algorithm has a complexity of

***O(10\*n)***. The "complexity" of his algorithm was much better than the first used.

We call this **linear complexity** because it increases **proportionally** to the number of digits.

**Complexity in constant time**

Solving the same problem of opening the briefcase, some one can come up with an idea to clash the padlock using a hammer, stone,...it seems so simple, now that we think about it...

If we take a closer look, its algorithm effective! **Regardless of the number of digits, it always takes the same time.**

We then speak of **constant time complexity** . We note this type of complexity with the **big O** notation, ***O(1)***

2.2.

1. **IMPLEMENT LINEAR DATA STRUCTURES**
2. **IMPLEMENT NON-LINEAR DATA STRUCTURES**
3. **IMPLEMENT HASH TABLE AND HANDLE DATA RETRIEVAL**
4. **APPLY RECURSION FOR SOLVING COMPLEX PROBLEMS USING C++**