LAB 01 TASKS

Question #01) Print the below patterns?

****

****

*****

**

Xx

Xxxxxxxx

Xxxxxxxxxxx

Source Code:

```cpp
#include <iostream>

using namespace std;

int main() {
    cout << "**********\n"
            "**********\n"
            "*****\n"
            "**\n"
            "Xx\n"
            "Xxxxxxxx\n"
            "Xxxxxxxxxxx";

    return 0;
}
```

Ouput:

```
**********
**********
*****
**
Xx
Xxxxxxxx
Xxxxxxxxxxx
--------------------------------
Process exited after 0.04453 seconds with return value 0
Press any key to continue . . .
```

Question #02) Make your CV that include your Name, Father's Name, CNIC, Qualification, Semester, CGPA etc.? Print each line separately using "\n" and "endl"? Use comments also.

Source Code:

```cpp
#include <iostream>
using namespace std;

int main() {
    // Personal Information
    cout << "Name: John Doe\n";
    cout << "Father's Name: Richard Doe\n";
    cout << "CNIC: 12345-56789012-3\n";

    // Academic Information
    cout << "Qualification: Undergraduate" << endl;
    cout << "Semester: 4" << endl;
    cout << "CGPA: 3.75" << endl;

    return 0;
}
```

Ouput:

```
Name: John Doe
Father's Name: Richard Doe
CNIC: 12345-56789012-3
Qualification: Undergraduate
Semester: 4
CGPA: 3.75

--------------------------------
Process exited after 0.03196 seconds with return value 0
Press any key to continue . . .
```

Question #03) Prepare list of available header files and their respective functions(Memorize them)

1. `<iostream>`

- **Functions**:
  - `std::cout`: Output stream for printing to the console.
  - `std::cin`: Input stream for reading from the console.
  - `std::clog`: Output stream for logging messages.
  - `std::cerr`: Output stream for errors.

2. **`<iomanip>`**

- **Functions**:
  - `std::setprecision()`: Sets the decimal precision for floating-point output.
  - `std::setw()`: Sets the width of the next output field.
  - `std::fixed`: Formats floating-point numbers in fixed-point notation.
  - `std::scientific`: Formats floating-point numbers in scientific notation.

3. **`<string>`**

- **Functions**:
  - `std::string`: Class for manipulating strings.
  - `std::getline()`: Reads a line from an input stream into a string.
  - `string::size()`: Returns the length of the string.
  - `string::substr()`: Returns a substring.

4. **`<vector>`**

- **Functions**:
  - `std::vector`: Dynamic array class.
  - `vector::push_back()`: Adds an element to the end.
  - `vector::pop_back()`: Removes the last element.
  - `vector::size()`: Returns the number of elements.

5. **`<algorithm>`**

- **Functions**:
  - `std::sort()`: Sorts a range of elements.
  - `std::reverse()`: Reverses the order of elements.
  - `std::find()`: Finds an element in a range.
  - `std::accumulate()`: Computes the sum of elements.

6. **`<cmath>`**

- **Functions**:
  - `std::sqrt()`: Returns the square root.
  - `std::pow()`: Raises a number to a power.
  - `std::sin()`, `std::cos()`, `std::tan()`: Trigonometric functions.
  - `std::log()`: Computes the natural logarithm.

7. **`<cstdlib>`**

- **Functions**:
  - `std::atoi()`: Converts a string to an integer.
  - `std::atof()`: Converts a string to a float.
  - `std::rand()`: Generates a random number.
  - `std::srand()`: Seeds the random number generator.

8. **`<ctime>`**

- **Functions**:
  - `std::time()`: Returns the current time.
  - `std::difftime()`: Computes the difference between two time values.
  - `std::ctime()`: Converts time to a string representation.

9. **`<fstream>`**

- **Functions**:
  - `std::ifstream`: Input file stream for reading from files.
  - `std::ofstream`: Output file stream for writing to files.
  - `std::fstream`: Input/Output file stream for both reading and writing.
  - `fstream::open()`: Opens a file.
  - `fstream::close()`: Closes a file.

10. **`<map>`**

- **Functions**:
  - `std::map`: Associative container that contains key-value pairs.
  - `map::insert()`: Inserts an element.
  - `map::find()`: Finds an element by key.
  - `map::erase()`: Removes an element by key.

11. **`<set>`**

- **Functions**:
  - `std::set`: Container that stores unique elements.
  - `set::insert()`: Inserts an element.
  - `set::find()`: Finds an element.
  - `set::erase()`: Removes an element.

12. **`<unordered_map>`**

- **Functions**:
  - `std::unordered_map`: Unordered associative container that contains key-value pairs.
  - `unordered_map::insert()`: Inserts an element.
  - `unordered_map::find()`: Finds an element by key.
  - `unordered_map::erase()`: Removes an element by key.

13. **`<thread>`**

- **Functions**:
  - `std::thread`: Class for creating and managing threads.
  - `thread::join()`: Waits for a thread to finish.
  - `thread::detach()`: Detaches a thread from the main thread.

14. **`<mutex>`**

- **Functions**:
  - `std::mutex`: Class for mutual exclusion to prevent data races.
  - `mutex::lock()`: Locks the mutex.
  - `mutex::unlock()`: Unlocks the mutex.

15. **`<condition_variable>`**

- **Functions**:
  - `std::condition_variable`: Class for blocking a thread until notified.
  - `condition_variable::wait()`: Blocks the current thread.
  - `condition_variable::notify_one()`: Unblocks one waiting thread.
  - `condition_variable::notify_all()`: Unblocks all waiting threads.

This list covers some of the most commonly used header files in C++. Each header file serves specific purposes and provides functions and classes that facilitate various programming tasks.

Question #04) Describe phases of compilation.

The compilation process in C++ consists of several distinct phases, each of which transforms the source code into an executable program. Here's a breakdown of the main phases:

1. Preprocessing

- **Description**: This is the first phase where the preprocessor handles directives (commands) that begin with `#`, such as `#include`, `#define`, and `#ifdef`.
- **Actions**:
  - **File Inclusion**: Replaces `#include` directives with the contents of the specified files (header files).
  - **Macro Expansion**: Expands macros defined by `#define`.
  - **Conditional Compilation**: Evaluates conditional directives and includes or excludes portions of code based on specified conditions.

2. Compilation

- **Description**: In this phase, the preprocessed code is translated into assembly code specific to the target architecture.
- **Actions**:
  - **Syntax and Semantic Analysis**: Checks the code for syntax errors and verifies that it follows the rules of the language.
  - **Intermediate Code Generation**: Converts the high-level code into an intermediate representation, which is easier to manipulate.
  - **Optimization**: May apply optimizations to improve performance and reduce resource usage (though this may vary depending on the compiler settings).

3. Assembly

- **Description**: The assembly code generated during the compilation phase is translated into machine code (object code).
- **Actions**:
  - The assembler converts the assembly language instructions into binary format that the computer's processor can execute.
  - This results in an object file, typically with a `.o` or `.obj` extension, containing machine code but not yet a complete executable.

4. Linking

- **Description**: The final phase where the object files generated from multiple source files are combined to create the final executable program.
- **Actions**:
  - **Symbol Resolution**: Resolves references to functions and variables between different object files.
  - **Library Linking**: Links against system libraries or other libraries specified by the programmer (e.g., standard libraries).
  - **Executable Generation**: Produces a final executable file (usually with an `.exe` extension on Windows or no extension on Unix-like systems).

Question #05) Describe different types of computer languages (at least 8).

1. High-Level Languages

- **Description**: These languages are designed to be easy for humans to read and write. They abstract away the complexities of the underlying hardware.
- **Examples**: Python, Java, C++, and Ruby.
- **Features**: Use of natural language elements, strong abstraction, and platform independence.

2. Low-Level Languages

- **Description**: Low-level languages provide little abstraction from a computer's instruction set architecture. They are closely related to machine code.
- **Examples**: Assembly language.
- **Features**: Direct manipulation of hardware, higher performance, and greater control, but more complex and less portable.

3. Machine Language

- **Description**: This is the lowest level of programming language, consisting of binary code that the computer's CPU can execute directly.
- **Examples**: Binary code (e.g., 01010100).
- **Features**: Fast execution and no need for translation, but difficult for humans to read and write.

4. Scripting Languages

- **Description**: Scripting languages are typically interpreted languages used for automating tasks and integrating systems.
- **Examples**: JavaScript, Python, Perl, and Bash.
- **Features**: Dynamic typing, ease of use, and often built for specific environments (like web browsers).

5. Functional Languages

- **Description**: These languages treat computation as the evaluation of mathematical functions and avoid changing state or mutable data.
- **Examples**: Haskell, Lisp, and Erlang.
- **Features**: First-class functions, immutability, and a focus on function application.

6. Object-Oriented Languages

- **Description**: These languages are based on the concept of "objects," which can contain data and code that manipulates that data.
- **Examples**: Java, C++, and C#.
- **Features**: Encapsulation, inheritance, and polymorphism, which help organize code and promote reuse.

7. Markup Languages

- **Description**: Markup languages are designed for the presentation of data rather than for computation.
- **Examples**: HTML (HyperText Markup Language), XML (eXtensible Markup Language), and Markdown.
- **Features**: Use of tags to define structure, primarily used for document formatting and data interchange.

8. Domain-Specific Languages (DSL)

- **Description**: DSLs are specialized languages designed for a specific domain or problem space.
- **Examples**: SQL (Structured Query Language for databases), VHDL (for hardware description), and CSS (Cascading Style Sheets for web design).
- **Features**: Tailored syntax and semantics that cater specifically to particular types of tasks.

9. Procedural Languages

- **Description**: These languages follow a set of procedures or routines to perform tasks.
- **Examples**: C, Fortran, and Pascal.
- **Features**: Emphasize a sequence of actions or commands, using constructs like loops, conditionals, and subroutines.

10. Concurrent Languages

- **Description**: Designed for concurrent programming, allowing multiple processes to run simultaneously.
- **Examples**: Go, Erlang, and Ada.
- **Features**: Built-in support for multi-threading and asynchronous operations