C++ Exercises
Set 1

Author(s): Olivier Gelling, Leon Lan, Mohammad Habibi

–

15:43

September 8, 2025

1

```
Exercise 1

We must write a hello world program, compile it into an object, or a pre-
compiled file, and then link that object into an executable.

The object (hello.o) file serves as a compiled version of this file without
being turned into a full working executable program. This makes it easier
to work with large programs where multiple source files are used. Changes
made to a single source file then do not require a complete recompilation of
the entire program, instead the specific source file can be recompiled, and
the rest of the (already) "pre-"compiled files can simply be linked by the
compiler. This saves a lot of time and computation power, and makes working
on large projects much more overseeable. It also ties in well with separate
file structures that often come with classes and such.

The executable file on the other hand is the complete set of functional
instructions that the computer must follow to run the program. It

The commands used for this exercise:
g++ -c hello.cc -Wall -Werror --std=c++26
to compile it, and
g++ hello.o -o hello -s -Wall -Werror --std=c++26
to link it

The sizes of our files in bytes:
hello.cc:   256
hello.o:    1632
hello:      14376
iostream:   3074
```

Listing 1: hello.cc

```
#include <iostream>                         // importing in-out-stream for cout

using namespace std;                        // adding namespace for ease

int main()                                  // args not needed explicitly
{
    cout << "Hello World" << '\n';
}
```

```
Hello World
```

2

```
Exercise 2

This question consists of multiple subquestions
```

Q1– In C++, what is the difference between a declaration and a definition?
A – A declaration provides identifiers, like for classes or functions, which can be referred to in other files. A declaration does not contain a code implementation yet. Definitions, on the other hand, actually specify the declared symbols in full, for example, giving functions a body, or variables a value.

Q2– What are header files used for?
A – Header files are used to provide declarations in the encompanying files. They serve as an overview of the usable parts of that they "head" so that their contents can be referred to elsewhere easily.

Q3– When does a compiler use a header file, and when a library?
A1– Header files – The compiler uses the header files in the preprocessing phase. It does so when preprocessing a source file, where any #include <header> directive is replaced with the contents of the header file.

A2– Libraries – These will be used when the compiler links the object files together into a working executable program. Until this point object files contain unresolved references. The linking phase resolves these references by linking the object files against the relevant libraries.

Q4– Is a library an object module? Explain as open question
A4– An object module is the product of an assembler or compiler, whereas a library contains object modules, but also header files, as such it is not simply an object module.

Q5– Why is an object file compiled from a source containing int main() not an executable program?
A5– An object file is not compiled as an executable program on its own because it is not linked and as such does not have its external references resolved that otherwise would be in the linking phase. It will thus not be able to run.

3

Exercise 3: Describe six major differences between c and c++

1 – const usage
Like C++, C has the const keyword, but its function and uses are greatly extended, as it now works on member functions, objects, and references, and functions can even be overloaded with const. Const is more rigorously enforced by the compiler with stricter rules, and embedded in its standard libraries.

2 – Namespaces
C++ allows for the use of namespaces, which C does not have. Namespaces allow for defining symbols in a specific context, without running into name conflicts.

3 – Scope resolution operator ::
The scope operator is newly introduced for C++, and allows for distinguishing between global and local variables with the same name. It is used to call class-bound function members or specify which namespace is being drawn from.

4 – cout, cin, cerr
C++ uses in and output streams similar to C's functions stdout, stdin, and stderr, but in C++ these are now stream objects (cout, cin, and cerr), rather than functions (like printf) that encompany file pointers.

5 – Function overloading
Contrarily to C, C++ allows functions to be overloaded, meaning that multiple functions can have the same name as long as they take different arguments, either by type, by amount, or a combination of both. Even

```
    const can be used for overloading.

6 - Classes, objects, inheritance
    C++ allows for classes, objects, inheritance, and other functionalities.
    Many of these are connected to defining and using classes, or classes
    inheriting functionalities from others. C simply has no classes to begin
    with, so cannot allow for any of the complexer functionalities that build
    on them.

We recognise that the last two answers are not part of the covered topics,
but found them interesting enough to use. We can also add differences in
structs, RSL's, and general additions to C's grammar instead on the final
hand-in. We hope though that this isn't an issue for a theory bound question
like this one.
```

4

```
Exercise 4 information

We choose a special combination for the start and end parentheses, used to
denote the start and end of a raw string literal, to differentiate them from
the parentheses that show up in the contents of the string literal, which
would otherwise be interpreted as the ending sequence of our string literal.
We can pick any unique combination of 'd-char's with the parenthesis of at
most 16 characters long, where a 'd-char' is a character from the basic
source character set (excluding parentheses, backslash and spaces). This will
then only look for that combination of keys inbetween the ) and ".

example:
R"veryuniquesetofcharacters(  raw string here  )veryuniquesetofcharacters"

We used R"special(  ...  )special"

The rule would thus become:
Use special parentheses when the content of the raw string literal contains
parentheses. Select special parentheses that do not appear in the content of
the raw string literal.
```

Listing 2: exercise4.cc

```
#include <iostream>

using namespace std;

namespace {                             // anonymous namespace

char const wifiPattern[] =
R"special(^\s+Encryption key:(\w+)
^\s+Quality=(\d+)
^\s+E?SSID:"([[:print:]]+)"
^\s+ssid="([[:print:]]+)"
)special";                              // adding newline by placing this here
// We use R"special( and )special" to denote the start and end of the string
};                                      // end of namespace

int main()
{
    cout << wifiPattern;
}
```

```
^\s+Encryption key:(\w+)
^\s+Quality=(\d+)
^\s+E?SSID:"([[:print:]]+)"
^\s+ssid="([[:print:]]+)"
```

5

Exercise 5: Submit an overview of standard escape sequences in table format

Esc seq	Short description
\	The escape sequence (ES) initialiser. Is not actually a "sequence" by itself, but effects the 'escaping'
%	Can be used in the same way as \ for dates and newlines
\n	New line. Terminates current line.
\\	A backslash: \
\'	A single quote: '
\"	A double quote: "
\?	A question mark: ?
\a	An audible alert bell
\b	A backspace
\f	Form feed, or new page
\r	Carriage return, returns to beginning of current line
\t	Horizontal tab
\v	Vertical tab
\nnn	Arbitrary octal value, using base 8 rather than 10. Is converted ->
\o{n...}	-> to ASCII, so \125 becomes U, and \101 'A' etc
\0	Octal terminating null character
\xn...	Arbitrary hexadecimal value, works similarly to octals above.
\x{n...}	
\c	Conditional, implementation defined ES. Does not work for most versions.
\unnnn	Arbitrary unicode value, in UTF-16 encoding
\u{n...}	
\Unnnnnnnn	Arbitrary unicode character, in UTF-32 encoding
\N{NAME}	Arbitrary unicode character by character name

What happens if another character is written as an escape sequence?
Provide example using a cout statement.

We write some code to show how this works:

```
#include <iostream>

using namespace std;

int main()
{
    cout << "new line: " << '\n';                    // Reference
    cout << "random character: " << '\q' << '\n';   // Testing false ES
}
```

If we try to write an escape sequence with a character for which no escape sequence is known to the compiler, like \q, it will throw a warning, like so:

```
exercise5.cc:8:37: warning: unknown escape sequence:    \ q
    8 |     cout << "random character :" << '\q';
      |                                      ^~~~
```

This will still compile if we do not have -Werror set though. if we run it we get:

```
new line:
random character: q
```

It seems that the escape sequence is annulled, and the character is simply read as is. We have added more tests to the encompanying exercise5.cc source file to see how specific escape sequences work.

6

Exercise 6

We must use all the different operators ONCE to check if a number is even or odd 6 times. We base this heavily off of the fact that true and false are also represented as 1 (or any nonzero value) and 0 respectively , to simplify the code.

Our first method simply uses modulo 2 to check if the answer is 0 (false/even) or 1 (true/odd).

Our second uses the bitwise OR operator to compare the LSB to the number 1. If the LSB of our input value is the same we have an odd number, else we are odd. This gives us a clear even and odd check.

Our third bitshifts the number one step to the right and then back to the left and compares it to the initial input value. If they are different the LSB was 1 and was replaced by a 0 due to the bitshift back.

Our fourth divides by two and then multiplies by two, and then checks if the number has changed from the input. If the input was even the resulting value should be the same. Otherwise integer division truncates the intermediate value towards 0, losing the decimal. If we then multiply it by two again we end up with a different value.

Our fifth uses bitwise OR to see if the LSB stays the same (if it was a 1) or if it flips (when it is 0). If the new value is the same as the input we have an odd number, else even.

Our sixth uses bitwise XOR with 1 to see whether the bit goes from 0 to 1, or from 1 to 0. If the new value is our input + 1 we had an even number, as the LSB was 0, otherwise it was odd, as the LSB was then 1 and got set to 0.

Listing 3: exercise6.cc

```
#include <iostream>

using namespace std;

int main()
{
    size_t value;
    cin >> value;

    cout << (value % 2 ? "odd" : "even") << '\n';
    // Simple modulo 2. 0 gives false, nonzero true

    cout << (value & 1 ? "odd" : "even") << '\n';
    // bitwise AND to select the LSB. 1/true for odd, 0/false for even

    cout << ((value >> 1) << 1 != value ? "odd" : "even") << '\n';
    // bitshift right and left to see if LSB goes from 1 to 0.

    cout << ((value / 2) * 2 != value ? "odd" : "even") << '\n';
    // divide and multiply by 2 to see if uneven numbers lose their .5

    cout << ((value | 1) == value ? "odd" : "even") << '\n';
    // use bitwise OR to check if last bit flips

    cout << ((value ^ 1) + 1 == value ? "odd" : "even") << '\n';
    // use bitwise XOR to see if the last bit flips up or down
}
```

7

Exercise 7:

Using command line arguments: Write single expression a program that outputs "hello" along with a specific command line argument or "world"

We use two nested ternary statements to distinguish between the three versions of the program's output:
1 – only the program is called. We want to output simply "hello world" here
2 – one argument is passed along. We want to output "hello [argument]'
3 – multiple arguments are passed along. We want to use the first argument as
    an index for which of the arguments is printed after "hello".

The first ternary operator statement filters out argc == 1; only the program is called and no command line arguments are. The "else" clause then gets another nested ternary operator statement to distinguish between argc == 2, which means one argument is passed along, and argc > 2.

We mimic exercise 6's format somewhat, where the ternary statements are nested inside cout, inserting whatever is selected due to the arg count after "hello ". This way we don't get errors concatenating the raw string with *argv[]'s character pointers.

Selecting argv[1] to insert into cout is then not very hard, but we need, as the hint shows, stoul() (string to unsigned long) to turn argv[1]'s input into a usable index for argv[] for the final output case.

Noteworthy is when an index larger than the argument list length is given. The first argument beyond the given list will simply be empty (and no endline will be registered), then it starts printing path variables, like SHELL, SESSION_MANAGER, etc. I am not sure if this is an error in my program, or a general issue with writing a program like this without any safeguards for the index argument being longer than argv[]

If we want the program to display the first argument while still handing it more than one we can set the first argument to be "1". This will cause it to output "hello 1". It seems that this also works with any first argument that starts with a "1" too. If we call "./a.out 1ooh lala chachacha" our output is "hello 1ooh"

Listing 4: exercise7.cc

```
#include <iostream>                       // importing relevant libraries
#include <string>

using namespace std;                      // namespace for ease

int main(int argc, char *argv[])          // using argc and *argv[] now
{
    cout << "hello " <<                   // we use insertion to avoid
    (                                     // concatenation issues
        argc == 1 ?                       // ternary check of argument count
        "world" :                         // set the second part of output
        (
            argc == 2 ?                   // second ternary check for 2 or more
            argv[1] :
            argv[stoul(argv[1])]          // cast argv[1] to usable index value
        )                                 // then select that index from argv[]
    )
    << '\n';
}
```

8

Exercise 8:

Write a program that takes 4 command line arguments as ip4 address octets and a fifth to define the netmask.

We define a single size_t and initialise it by taking the four commandline octet values, casting them to unsigned longs with stoul, and then shifting their bit values up the size_t. The first octet gets shifted 24 bits to the left, so as to take the leftmost 8 bits (32 − 24), the second gets shifted 16, the third 8, and the fourth none. This way the size_t's bits are divided into 4 sections of 8 bits that each store an octet.

To do this we use bitwise OR: |. This operator lets us combine these octets after shifting them easily, by comparing the bits. It keeps the already stored values and is a useful and safe way to combine data in this way.

Noteworthy is that due to our expectation that input values will be correct we can just use + here, but it seems that | is the norm for actions like these. As such we have decided to use this format.

We apply the mask right away. We use ˜0UL, which in practice is a bitstring with all the bits set to 1. We shift this mask to the left by (32 − 'common bits') where common bits is the netmask given as fifth commandline argument. This way there will be 0s on the rightmost bits. We then use bitwise AND/& to filter out only the relevant values from the IP address, the ones where the mask is 0 will be suppressed.

We do the a similar thing when we want to read the octets again in cout. We shift the octets back to the right so that they can be read as the lowest 8 bits. we then apply the mask '& 255'. Which is a bitstring of eight 1's. This allows us to read only the current rightmost 8 bits, and thus keeps us from getting large numbers for the outputs of the three righter octets.

The outputs with 20 and 16 common bits are the same because the third octet is simply the number 3. It will have been cut off by the time only 20 bits are read. If the third octet had been a longer nuber more of it would have survived the 20 bit mask.

Listing 5: exercise8.cc

```cpp
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    size_t ip4 =
    (
        (stoul(argv[1]) << 24) |                // Setting first octet
        (stoul(argv[2]) << 16) |                // second octet
        (stoul(argv[3]) << 8) |                 // third
        stoul(argv[4])                          // fourth
    )
    & ~0UL << (32 - stoul(argv[5]));            // apply mask

    cout << ((ip4 >> 24)) << '.'                // read first octet
         << ((ip4 >> 16) & 255) << '.'          // read second
         << ((ip4 >> 8) & 255) << '.'           // read third
         << (ip4 & 255)                         // read fourth
         << '\n';                               // We mask them with '255'
                                                // to only read those 8 bits
}
```

```
129.125.0.0

129.125.0.0

129.125.3.0

129.125.3.160

129.125.3.162
```