Module 9: C++ Lecture Exercises

Each section in the Lecture Exercises should be completed after watching the corresponding lecture section.

Part 1: First C++ Program and Compiler Errors

1. Hello, Student! 🌍

Motivation

Your first C++ program introduces you to the basic structure of C++ code and helps you understand the compilation and execution process as discussed in the lecture.

Preparation

Make sure your C++ development environment is properly set up. Review the basic structure of a C++ program from the lecture, including the `#include` directive, the `main()` function, and how to use `std::cout` for output.

Exercise

Write a simple C++ program that prints **"Hello, YOURNAME!"** to the console. Compile and run it using the instructions from the lecture.

---

2. Fixing Compilation Errors 🛠️

Motivation

Understanding and fixing syntax errors is an essential skill for any programmer. As we saw in the lecture, C++ catches errors during compilation rather than execution. This exercise helps you recognize common C++ syntax issues and teaches you how to interpret compiler error messages.

Preparation

Review the compiler error messages discussed in the lecture and the common syntax rules of C++, such as the need for semicolons at the end of statements, which was emphasized in our discussion of C++ syntax.

Exercise

This program has **syntax errors**. Fix them so that it compiles and runs correctly.

```C/C++
#include <iostream>

int main() {
    int x = 5
    std::cout < "The value of x is: " << x << std::endl
    return 0;
}
```

Once fixed, modify the program to **print the square of `x`**.

Part 2: C++ Data Types

3. Completing the Data Types Table 📝

Motivation

Understanding data types and their memory requirements is fundamental in C++. As we explored in the lecture, C++ requires explicit declaration of types, and each data type uses a specific amount of memory. This exercise will help you grasp how different data types use memory and the range of values they can represent.

Preparation

Review the different C++ data types covered in the lecture, particularly the fundamental types table. Recall that C++ has specific guaranteed minimum sizes for each type. However, this is a guaranteed minimum, it may be larger on your computer.

Remember how we calculated the range of integer values in the lecture:

- For a signed integer type with n bits, the range is from -2^(n-1) to 2^(n-1)-1
- For example, a 16-bit signed integer (like `short int`) can represent values from -2^15 to 2^15-1, which is -32,768 to 32,767
- We use 2 as the base because each bit in binary can have one of two values (0 or 1)
- We use n-1 for signed integers because one bit is used to represent the sign (positive or negative)

For this exercise, you'll need to use the `sizeof()` operator, which wasn't covered in the lecture. The `sizeof()` operator is a built-in C++ operator that returns the size in bytes of a data type or variable. Here's how to use it:

```
C/C++
// Get size of a type
int size_of_short_int = sizeof(short int);  // Use std::cout to print value

// Get size of a variable
short int x = 10;
int size_of_x = sizeof(x);
```

When used with a type name (like `short int` or `double`), the type name must be enclosed in parentheses. When used with a variable name, the parentheses are optional but recommended for consistency.

Remember: 1 byte = 8 bits, so a 2-byte short integer has 16 bits total.

Exercise

Complete the following table using the lecture material and by writing a small program to check sizes using `sizeof()`.

Type	Min Size (bytes)	Measured Size (bytes)	Typical Range	Example
int				`int x = 42;`
long int				`long int y = 1000000;`
short int				`short int s = 100;`
long long				`long long x = 10000;`
float			X	`float pi = 3.14159;`
double			X	`double precise = 3.141592653589793;`

Note:

1. You do not need to fill in the "Typical Range" for float and double types (marked with X). The representation of floating-point ranges is complex and requires scientific notation, which is beyond the scope of this exercise.
2. All integer types in this table (`int`, `long int`, `short int`) are signed by default in C++. This means they can represent both positive and negative values.

---

4. Integer Overflow Experiment 🔥

Motivation

Integer overflow is a common source of bugs in C++ programs. As we discussed in the lecture, the range of possible values for an integer is determined by the number of bits it uses. Understanding how and why overflow occurs will help avoid unexpected behavior in numerical computations.

Preparation

Recall from the lecture that an integer type can only represent a finite range of values. For example, a 16-bit `short int` typically ranges from -32,768 to 32,767. Consider what happens

when a calculation exceeds the maximum or minimum value that can be stored in an integer type.

Exercise

Using your **data types table** from Exercise 3, determine the maximum and minimum values for a `short int`. Write a program that:

1. Assigns the **largest possible `short int`** to a variable.
2. Adds 1 to that variable and prints the result.
3. Assigns the **smallest possible `short int`** to another variable.
4. Subtracts 1 from that variable and prints the result.

Before running the program, **predict what will happen**.
After running it, **explain the behavior you observe**.

Part 3: Python and C++ Comparison

5. Performance Comparison 🕑

Motivation

Different programming languages offer different performance characteristics. As highlighted in the lecture, C++ is typically faster than Python for computational tasks because it is compiled directly to machine code, while Python is interpreted at runtime. This exercise allows you to observe the performance gap.

Preparation

Think about the execution models of Python versus C++ that we discussed. Consider why compiled languages might offer better performance than interpreted languages for computationally intensive tasks.

Timing Code

You'll need to measure how long your program takes to run in each language.

In Python, we can use the time library.

```python
import time

start_time = time.time()

# Your summation code here

end_time = time.time()
print(f"Execution time: {end_time - start_time} seconds")
```

In C++, we can use the chrono library.

```cpp
#include <iostream>
#include <chrono>

int main() {
```

```
    auto start_time = std::chrono::steady_clock::now();

    // Your summation code here

    auto end_time = std::chrono::steady_clock::now();
    std::chrono::duration<double> elapsed = end_time - start_time;
    std::cout << "Execution time: " << elapsed.count() << " seconds" <<
std::endl;

    return 0;
}
```

Exercise

1. Write a Python program that sums the numbers from `0` to `100,000,000` (**not including** `100,000,000`) using a loop. Include the timing code above.

2. Write a C++ program that does the same. You may need to look up the syntax for a `for` loop if it's unfamiliar.

    - Before writing the loop consider what integer type you will need to use for your sum. Recall the minimum and maximum values for types you calculated in the problems above.
    - We haven't covered the syntax for a for loop in C++, here is an example that counts to 5.

```
C/C++
for(int i=0; i<5; i++)
{
   std::cout << i << std::endl;
}
```

3. Run both programs on your machine and record the execution time for each. Which version ran faster? By approximately what factor?

---

6 . When do the errors occur? ❌

Motivation

Understanding when errors are detected is a fundamental difference between compiled languages like C++ and interpreted languages like Python. As we observed in the lecture, these different execution models have significant implications.

Preparation

Recall the execution process for both Python and C++ discussed in the lecture. Remember that Python's interpreter translates code line-by-line during execution, while C++ compiles the entire program before execution begins.

Exercise

Intentionally add an error to your code from (5) to the last line (print for Python, or cout for C++). When you attempt to execute or (compile and execute), how long does it take for the error to be detected?