Practical 6

COS132

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science

Official Deadline: 03/05/2024 at 17:30

Extended Deadline: 05/05/2024 at 23:59

Marks: 180

1 General instructions:

- This assignment should be completed individually; no group effort is allowed.
- Be ready to upload your practical well before the deadline, as no extension will be granted.
- **The extended deadline has been put in place in case of loadshedding or other unforeseen circumstances. No further extension will be granted.**
- **Note that no tutor or lecturer will be available after the official deadline**
- You may not import any libraries that have not been imported for you, except `<iostream>`.
- You may `use namespace std;`
- If your code does not compile, you will be awarded a zero mark. Only the output of your program will be considered for marks, but your code may be inspected for the presence or absence of certain prescribed features.
- All submissions will be checked for plagiarism.
- Read the entire practical before you start coding.
- You will be afforded 20 upload opportunities per task.

- **You should no longer be using the online compiler. You should compile, run and test your code locally on your machine using terminal commands or makefiles**.
- **You have to use C++98 in order to get marks**

2 Plagiarism

The Department of Computer Science considers plagiarism a serious offence. Disciplinary action will be taken against students who commit plagiarism. Plagiarism includes copying someone else's work without consent, copying a friend's work (even with permission) and copying material from the Internet. Copying will not be tolerated in this course. For a formal definition of plagiarism, the student is referred to http://www.library.up.ac.za/plagiarism/index.htm. **If you have any questions regarding this, please ask one of the lecturers to avoid misunderstanding.**

3 Outcomes

Upon successful completion of this practical, students will be able to:

- Understand and apply basic control structures in C++, including if-else statements, switch statements, and for loops, to solve programming problems.
- Identify and correct common syntactic and semantic errors in C++ code, improving their debugging skills and code comprehension.
- Utilize logical reasoning to rearrange and fix nested control structures to ensure that the program behaves as intended.
- Develop a systematic approach to testing their code by writing main functions that test different cases for parameter input

4 Structure

This practical consists of two tasks. Each task is self-contained and all of the code you will require to complete it will be provided in the appropriate Task folder. Each task will require you to submit a separate archive to an individual upload slot. That is, each separate task will require its own answer archive upload. You will upload Task 1 to Practical 6 Task 1 and so on. You can assume that all input will be valid (i.e. of the correct data type and within range)

5 Mark Distribution

Activity	**Mark**
Task 1 - Switch, If and For	50
Task 2 - Codebreaker	130
Total	**180**

Table 1: Mark Distribution

6 Resources

Here are some additional resources you can use to help you complete the practical

- **if statement**: The `if` statement is used to execute a block of code only if a specified condition is true. The syntax is

  ```
  if (condition) {
      statement(s)
  }
  ```

  If the condition evaluates to true, the statements inside the curly braces are executed. If the condition is false, the code block is skipped. More details can be found here: `https://www.geeksforgeeks.org/c-c-if-statement-with-examples/`.

- **else if statement**: The `else if` statement, which follows an `if` statement, allows for multiple sequential checks. It is used to specify a new condition to test, if the previous condition was false. The syntax is

  ```
  if (condition1) {
      statement(s)
  } else if (condition2) {
      otherStatement(s)
  }
  ```

  Each `else if` is checked in order. If a condition evaluates to true, the corresponding statements are executed, and the rest of the `else if` chain is bypassed. More on this can be read at: `https://www.geeksforgeeks.org/c-c-if-else-statement-with-examples/`.

- **else statement**: The `else` statement, which follows an `if`, or sequence of `if`, `else if` statements, allows for a defult behaviour when none of the conditions are met. It is used to specify a behaviour if all condition checks failed. The syntax is

  ```
  if (condition1) {
      statement(s)
  } else {
      defaultStatements(s)
  ```

```
}
```

The `else` is only executed if no `if` or `else if` in the chain evaluated to true. More on this can be read at: `https://www.geeksforgeeks.org/c-c-if-else-statement-with-examples/`.

- **switch statement**: The `switch` statement is a control structure used to perform different actions based on different conditions. It's an alternative to multiple `if` statements and is generally used to simplify complex decision-making processes. The syntax is

  ```
  switch(expression) {
      case constant1:
              statement(s);
              break;
      case constant2:
              statement(s);
              break;
      //Other cases
      default:
              statement(s);
  }
  ```

  The expression is evaluated once, and the execution jumps to the `case` that matches the result of the expression, continuing until a `break` is encountered or the end of the `switch` block. More details can be found here: `https://www.geeksforgeeks.org/switch-statement-in-cpp/`.

- **for loop**: The `for` loop is used to repeat a section of code a **known** number of times. It is often used when the number of iterations is known beforehand. The syntax for the `for` loop is

  ```
  for(initialization; condition; increment) {
      statement(s);
  }
  ```

  The loop starts with the initialization step, followed by checking the condition. If the condition is true, the loop body is executed, followed by the increment step. This process repeats until the condition evaluates to false. For further details and examples, visit: `https://www.geeksforgeeks.org/cpp-for-loop/?ref=lbp`.

7 Task 1

Your task is to fix the semantic and sytnax errors of the if, switch and for loops we provided. **Do not change ANY of the print statements. You can move them, delete them or duplicate them if necessary, but do not change them**

7.1 Instructions for Correcting If Statement Challenges

1. **void checkEvenOrOdd(int number)**
   - This function should print whether the passed in parameter is positive even, positive odd, 0 or negative.
   - Fix the nesting so that this is achieved.
   - Do not change the actual print statements, just move them or add brackets as required.
2. **checkWhereXRanks(int x, int y, int z)**
   - This function takes in x, y, and z.
   - You are required to print whether x is the greatest, the smallest or neither
   - Rearrange or update the conditions and the print statements so that they are logical. Eg. x = 2; y = 2; z = 5; Should print "X is neither the smallest nor the greatest"
   - You may add, delete, or modify conditions. Just do not modify print statements
3. **checkInRange(int x)**
   - Verifies that x is between 1 and 9 inclusive.
   - If it is not - do not print anything

Note: Ensure each conditional expression is clear and unambiguous to maintain the intended logic of the program. Test each segment individually to verify correctness.

7.2 Instructions for Correcting Switch Statement Challenges

1. **void testDayOfWeek(int day)**
   - This function should print the day of the week corresponding to the parameter
     - 1 –> Monday
     - 2 –> Tuesday
     - etc.
2. **switchRanges(int score);**
   - This function attempts to handle score ranges using a method that is typically not suited for range checking. Adjust it using an appropriate control structure that better handles continuous range evaluations. Ensure the output remains semantically correct.

Note: Each correction should be tested by compiling the code to ensure that all syntax errors are resolved and the logic behaves as expected under various conditions.

7.3 Instructions for Correcting For Loop Challenges

1. **testBasicFor(int upTo)**
   - This function should print the values of `i` from 0 until upTo (exclusive of upTo). Correct the error.
   - Do not change the print statement
2. **testDoubleLoopVar(int x, int y)**
   - This function should let `i` start `x` and `j` at `y`
   - They should move towards each other in values until `i <= j`.
   - For example, if x = 1 and y = 5:
     i: 1, j: 5 i: 2, j: 4 i: 3, j: 3
   - Do not change the print statements
3. **printOnlyEvenInLoop(int upTo)**
   - Print all even numbers from 0 to upTo (excluding upTo)
   - Do not change the print statement
   - Hint: You can add an `if` in a for loop
4. **printPowers(int x, int upTo)**
   - Print the powers of x, smaller than upTo, starting at x
   - Eg. For x = 2, and upTo = 35:
     i: 2 i: 4 i: 8 i: 16 i: 32

8 Task 2

You are a core member of "Freedom Bytes," a clandestine group of elite digital liberators and vigilantes. Recently, you've unearthed alarming evidence suggesting that a shadowy corporation known as "CodeBrew" has been covertly manipulating global markets and political outcomes. Using their extensive and highly secure data networks, CodeBrew's influence runs deep, affecting the very fabric of democracy and free trade worldwide.
Determined to expose and dismantle CodeBrew's operations, "Freedom Bytes" has initiated "Operation Codebreaker." This high-stakes mission is divided into three critical phases, each designed to infiltrate, destabilize, and ultimately cripple CodeBrew's manipulative data infrastructure.

8.1 Header and Source File Creation Guide

This guide outlines the structure and responsibilities of the header ('.h') and source ('.cpp') files for the monitoring system. Each header file contains the declarations of functions related to the environmental controls for temperature, pressure, and humidity.

8.1.1 Database Corruption Header

The `DatabaseCorruption.h` header file includes the declarations for functions that simulate corrupting a database based on security levels.

```
#ifndef DATABASE_CORRUPTION_H
#define DATABASE_CORRUPTION_H
#include <iostream>

void corruptDatabase(int securityLevel);
void initiateCorruptionSequence();
void finalizeCorruption();

#endif // DATABASE_CORRUPTION_H
```

Create `DatabaseCorruption.cpp` and include the `DatabaseCorruption.h` header.

```
#include "DatabaseCorruption.h"

// Function implementations for corrupting the database will go here
```

8.1.2 Database Search Header

The `DatabaseSearch.h` header file contains the function declarations related to searching a database and verifying file paths.

```
#ifndef DATABASE_SEARCH_H
#define DATABASE_SEARCH_H
#include <iostream>

int searchDirectory();
int findDatabaseFile(int fileIndex);
int verifyDatabasePath(int dbIndex);
bool isPrime(int num);

#endif // DATABASE_SEARCH_H
```

Create `DatabaseSearch.cpp` and include the `DatabaseSearch.h` header at the top:

```
#include "DatabaseSearch.h"

// Implement the functions to search the database and verify file paths
```

8.1.3 Network Infiltration Header and Implementation

The `NetworkInfiltration.h` file includes the necessary function declarations for simulating network security infiltration.

```
#ifndef NETWORK_INFILTRATION_H
#define NETWORK_INFILTRATION_H
#include <iostream>

static int layer = 1;
int infiltrateNetworkLayer( int bypassKeyA, int bypassKeyB);
int bypassFirewall(int firewallLevelA, int firewallLevelB);
int bypassEncryption(int encryptionLevelA, int encryptionLevelB);
int gcd(int a, int b);
int accessMainframe();
void setLayer(int l);

#endif // NETWORK_INFILTRATION_H
```

Cretae `NetworkInfiltration.cpp` and include the `NetworkInfiltration.h` header at the top:

```
#include "NetworkInfiltration.h"

// Implement network infiltration functions
```

Each '.cpp' file should implement the functions declared in its respective '.h' file. Pay attention to ensure that each '.cpp' file includes its corresponding header file to provide function prototypes, enabling proper compilation.

8.2 Detailed Function Implementation Specifications for Network Infiltration

This module simulates a complex process of bypassing network security layers using various techniques and keys.

8.2.1 Global Variables

- **Static Integer Layer:** Control the sequence of network security layers being infiltrated. Already initialised in header file

8.2.2 int infiltrateNetworkLayer(int bypassKeyA, int bypassKeyB)

This function serves as the central controller for navigating through different network security layers.

- **Parameters:**
  - **bypassKeyA:** The first key used for attempting to bypass security mechanisms.

- **bypassKeyB:** The second key used alongside bypassKeyA.

- **Process:**

  1. Use a switch statement on the global `layer` variable to determine the function to call:
     - **Case 1:** Calls `bypassFirewall(bypassKeyA, bypassKeyB)` and returns the result.
     - **Case 2:** Calls `bypassEncryption(bypassKeyA, bypassKeyB)` and returns the result.
     - **Case 3:** Calls `accessMainframe()` and returns the result.
     - **Default:** Prints "Layer not recognized" to indicate an invalid layer and return -1.

8.2.3 int bypassFirewall(int firewallLevelA, int firewallLevelB)

This function simulates the process of cracking a firewall using two distinct security levels.

- **Parameters:**
  - **firewallLevelA:** The complexity level of the first stage of the firewall.
  - **firewallLevelB:** The complexity level of the second stage of the firewall.

- **Process:**

  1. Initialization of Variables
     - Six integer variables are used to manage the Fibonacci-like sequence for both firewall levels:
       * **part1A, part1B, part1C:** These variables manage the sequence for Firewall Level A.
       * **part2A, part2B, part2C:** These variables are used for the sequence in Firewall Level B.
     - `part1A` and `part2A` are initialized to 0, and `part1B` and `part2B` are initialized to 1, which are the starting values for a Fibonacci sequence.
  2. Cracking Process
     (a) **Firewall Level A:**
        i. A `for` loop iterates from 0 up to `firewallLevelA` (inclusive). In each iteration, the Fibonacci-like sequence is updated:
           - `part1C` is calculated as the sum of `part1A` and `part1B`.
           - Print `part1C` followed by an endline, eg `cout«part1C«endl;`
           - The values of `part1A` and `part1B` are then updated for the next iteration.
        ii. Once the fibonacci sequence up to `firewallLevelA` has been calculated, it prints "Cracked Level A...\n", indicating that the first level of the firewall has been successfully cracked.

     (b) **Firewall Level B:**

    i. Following the completion of Level A, another `for` loop begins, iterating up to `firewallLevelB` (inclusive) with similar sequence updates:
       - `part2C` is calculated as the sum of `part2A` and `part2B`.
       - Print `part2C` followed by an endline, eg `cout«part2C«endl;`
       - The values of `part2A` and `part2B` are then updated for the next iteration.
    ii. Upon reaching the final iteration for `firewallLevelB`, it prints "Cracked Level B... Firewall cracked\n", signaling that the second level of the firewall has been breached.

(c) After successfully cracking both levels:
    i. increments `layer`, `firewallLevelA`, and `firewallLevelB`
    ii. Call `infiltrateNetworkLayer` with the updated parameters to proceed to the next layer of security and return the result.

8.2.4 int gcd(int a, int b)

- **Purpose:** The `gcd` function computes the greatest common divisor of two integers using the Euclidean algorithm, which is an efficient method for computing the GCD.
- **Parameters:**
  - **int a:** The first integer.
  - **int b:** The second integer.
- **Process:**
  1. If `b` equals 0, the algorithm returns `a`. This is the base case, as the GCD of a number and 0 is the number itself.
  2. If `b` is not 0, the function calls itself with `b` and `a % b` (the remainder of `a` divided by `b`). This step effectively reduces the size of the numbers, leveraging the property that the GCD of two numbers also divides their difference.
- **Example:**
  - Calculating the GCD of 48 and 18:
    1. `gcd(48, 18)` first computes `18` and `48 % 18` (which is 12).
    2. Then `gcd(18, 12)`, next `gcd(12, 6)`, and finally `gcd(6, 0)`, which returns 6.

8.2.5 int bypassEncryption(int encryptionLevelA, int encryptionLevelB)

This function aims to simulate decrypting encrypted data by determining the greatest common divisor (GCD) of two encryption keys. The GCD finds the largest number that divides both of two integers without leaving a remainder.

- **Parameters:**
  - **int encryptionLevelA:** Represents the complexity or level of one part of the encryption.

  - **int encryptionLevelB:** Represents the complexity or level of another part of the encryption.

- **Process:**

  1. The function calls `gcd` with `encryptionLevelA` and `encryptionLevelB` as arguments.
  2. It stores the result in `result`, which represents the computed GCD of the two encryption levels.
  3. Print:
     `cout « "GCD of " « encryptionLevelA « " and " « encryptionLevelB « " is:  " « result « "\nPlain text password found\n";`
  4. Increments `layer`
  5. Call `infiltrateNetworkLayer` with `encryptionLevelA` and `encryptionLevelB` to continue to the next layer of security and returns the result.

8.2.6 int accessMainframe()

This function represents achieving ultimate access to the network's central mainframe.

- **Process:**

  1. Prints "Accessing Mainframe...\n" indicating successful penetration into the network's most secure data center.
  2. Return 10

8.2.7 void setLayer(int l)

This functions sets layer equal to the parameter l, ie. `layer = l;`

8.3 Detailed Function Implementation Specifications for Database Search

This module simulates searching through a database directory, checking for file integrity, and identifying prime numbers within a specified range. It employs basic mathematical operations to demonstrate computational processes.

8.3.1 void searchDirectory()

This function calculates the highest factorial that does not exceed the maximum integer limit to simulate the potential for integer overflow in database operations.

- **Parameters:** None.

- **Process:**

  1. Initializes an integer `maxInt` to 2147483647 (the maximum value for a signed 32-bit integer) and a double `factorial` to 1 and an integer loop control variable `i` (you need to declare the loop control variable here as it will be needed after the `for` loop).

2. Uses a `for` loop to calculate factorials incrementally until the maximum value is reached without causing an overflow.
   (a) The loop control variable should start at 1
   (b) The condition should read: factorial <= maxInt / <loopControlVariable>
   (c) The update should increment the loop control variable
   (d) Inside the loop, update factorial as `factorial` multiplied by the loop control variable
3. Print the highest factorial index computed without overflow, with a newline at the end:
   `Highest factorial without overflow is: <loopControlVar-1> ! = <factorial>`
   replace <loopControlVar> and <factorial> with appropriate variables and make sure to add a newline at the end.
4. Call `findDatabaseFile(i)` and return the result.

8.3.2 bool isPrime(int num)

This function determines whether a given number is prime, useful in database operations for data validation or cryptographic applications.

- **Parameters:**
  - **int num:** The number to be checked for primality.
- **Process:**
  1. Checks if `num` is less than or equal to 1, immediately returning false as these are not prime numbers.
  2. Checks if `num` is less than 4 but greater than 1, returning true since 2 and 3 are prime numbers.
  3. Uses a `for` loop to find the remainder of `num` divided by every integer from 2 up to `num - 1` (inclusive). If any division results in no remainder, returns false.
  4. Returns true if no divisors are found, confirming that `num` is a prime number.

8.3.3 void findDatabaseFile(int maxNum)

This function iterates through a range up to `maxNum`, checking if each number is prime and performing database path verification for non-prime numbers.

- **Parameters:**
  - **int maxNum:** The upper limit for the numbers to be checked, derived from the previous factorial calculation.
- **Process:**
  1. Uses a `for` loop from 1 to `maxNum` (inclusive).

   (a) Calls `isPrime()` on each number. If true, prints the number with a message indicating it is a prime. `<loopControlVariable> is a prime number.\n`
   (b) If `isPrime()` returns false, call `verifyDatabasePath()` to simulate a check on the database path for non-prime numbers. If `verifyDatabasePath()` returns a non-zero value, return the value, else continue with the loop.

2. return 0;

8.3.4 int verifyDatabasePath(int dbIndex)

This function simulates the verification of a database path, typically used to ensure database integrity or to validate access permissions.

- **Process:**

  1. If dbIndex is equal to 12, then:
     – Print "Verifying database path...C://home//CEO_Code//DB\n" to simulate a path verification process.
     – Return 80
  2. Else:
     – Print "Verifying database path...Incorrect index\n"
     – Return 0

8.4 Detailed Function Implementation Specifications for Database Corruption

This module simulates the process of database corruption, conditioned on a security clearance level.

8.4.1 void corruptDatabase(int securityLevel)

This function initiates the database corruption process based on a given security level.

- **Parameters:**
  – **int securityLevel:** The security clearance level attempting to initiate database corruption.

- **Process:**

  1. Checks if the `securityLevel` is greater than 80.
  2. If true, calls `initiateCorruptionSequence()` to start the corruption process.
  3. If not, prints "Security level too low to initiate corruption." with a newline at the end. This serves as a check against unauthorized attempts to corrupt the database.

8.4.2 void initiateCorruptionSequence()

This function orchestrates the sequence of corruption activities to disrupt database integrity.

- **Process:**
  1. Uses a `for` loop to iterate `i` from 0 to 9 (inclusive), simulating ten steps of data corruption.
  2. Within the loop, a `switch` statement is used based on the modulus of `i` by 3 to vary the type of corruption:
     - **Case 0:** Injects corruption packets into the database at the current iteration index, printing "Injecting corruption packet [i]" with a newline at the end (where [i] is the value of `i`.
     - **Case 1:** Scrambles database entries, affecting data integrity directly, printing "Scrambling database entry [i]" with a newline at the end (where [i] is the value of `i`).
     - **Case 2:** Overloads data lines to disrupt data transmission, printing "Overloading data line [i]" with a newline at the end (where [i] is the value of `i`).
- After completing the loop, calls `finalizeCorruption()` to end the corruption process.

8.4.3 void finalizeCorruption()

This function concludes the corruption sequence and logs the outcome.

- **Process:**
  1. Prints "Database corruption complete. Data integrity compromised." with a newline at the end. This message signifies that the database has been fully corrupted and its data integrity is no longer reliable.

9 Submission checklist

For Task 1:

- Archive (zip) all the files used for Task 1 and rename the archive uXXXXXXXX.zip where XXXXXXX is your student number. The zip should include:
  - testIf.cpp
  - testSwitch.cpp
  - testFor.cpp
- Upload the archive to FitchFork Practical 6 Task 1 before the deadline

For Task 2:

- Archive (zip) all files used for Task 2 and rename the archive uXXXXXXXX.zip where XXXXXXX is your student number. The zip should include:

  - `DatabaseCorruption.h`
  - `DatabaseCorruption.cpp`
  - `DatabaseSearch.h`
  - `DatabaseSearch.cpp`
  - `NetworkInfiltration.h`
  - `NetworkInfiltration.cpp`
- Upload the archive to FitchFork Practical 6 Task 2 before the deadline