3 - Debugging C++ Programs

3.1 - Syntax and semantic errors

- A syntax error occurs when you write a statement that is not valid according to C++ grammar.
- A semantic error is when the script doesn't behave as it was intended to during creation.

```
#include <iostream>

int add(int x, int y)
{
    return x - y; // function is supposed to add, but it doesn't
}

int main()
{
    std::cout << add(5, 3); // should produce 8, but produces 2

    return 0;
}
```

3.2 - The debugging process

- Find the root cause
- Understand the problem
- Determine a fix
- Repair the issue
- Retest

3.4 - Basic debugging tactics

- Comment out code : find out the source of the error
- Validate code flow : use print statements at the top of functions to ensure that it's being called

```
int getValue()
{
std::cerr << "getValue() called\n";
  return 4;
}

int main()
{
std::cerr << "main() called\n";
    std::cout << getValue;

    return 0;
}
```

- Print values : print values returned by function
- But using print statements while debugging code is not the best option:
  - More non-reusable code is added
  - Unwanted code cluttering
  - Entails modifying code and can introduce new bugs

3.5 - More debugging tactics

Here better debugging tactics as opposed to using print statements are discussed

- **Preprocessor directives** : an easier way to enable and disable debugging in programs

```
#include <iostream>
#define ENABLE_DEBUG // comment out to disable debugging

int getUserInput()
{
#ifdef ENABLE_DEBUG
std::cerr << "getUserInput() called\n";
#endif
  std::cout << "Enter a number: ";
  int x;
  std::cin >> x;
  return x;
}

int main()
{
```

```
#ifdef ENABLE_DEBUG
std::cerr << "main() called\n";
#endif
    int x = getUserInput();
    std::cout << "You entered: " << x;

    return 0;
}
```

- **Using a logger** :

```
# include <iostream>
# include "plog/Log.h" // Step 1: include the logger header

int getUserInput()
{
    LOGD << "getUserInput() called"; // LOGD is defined by the plog library

    std::cout << "Enter a number einstien : ";
    int num{0};
    std::cin >> num;
    return num;
}

int main()
{
    plog::init(plog::debug. 'logfile.txt'); // Step 2: initialize the logger
    plog::init(plog::none , "Logfile.txt"); // plog::none eliminates writing of most
                                            //messages, essentially turning logging off

    LOGD << "main() called"; // Step 3: Output to the log as if you were writing to the console
    int x{getUserInput()};
    std::cout << "You entered : "x;
    return 0;
}
```

3.10 - Finding issues before they become problems

- Don't make errors
    - Follow best practices
    - Don’t program when tired
    - Understand where the common pitfalls are in a language (all those things we warn you not to do)
    - Keep your programs simple

    - Don’t let your functions get too long
    - Prefer using the standard library to writing your own code, when possible.
    - Comment liberally.
- Refactoring functions
    - One way to address this is to break a single long function into multiple shorter functions. This process of making structural changes to your code without changing its behavior (typically in order to make it more maintainable) is called refactoring.
- Defensive programming
    - A practice whereby the programmer tries to anticipate all of the ways the software could be misused, either by end-users, or by other developers (including the programmer themselves) using the code. These misuses can often be detected and then mitigated (e.g. by asking a user who entered bad input to try again)
- Testing Functions

```
#include <iostream>

int add(int x, int y)
{
  return x + y;
}

void testadd()
{
  std::cout << "This function should print: 2 0 0 -2\n";
  std::cout << add(1, 1) << " ";
  std::cout << add(-1, 1) << " ";
  std::cout << add(1, -1) << " ";
  std::cout << add(-1, -1) << " ";
}

int main()
{
  testadd();

  return 0;
}
```

- Constraints
    - Addition of extra code to ensure use inputs are in the expected format
    - Assert and static_assert. can be used to impose constraints
- Static analysis tools

- programs that analyze your code to identify specific semantic issues