

# CS23820 Assignment

## Fractal Colouring Book Generator

Ethan Swain  
220029553 – ets6

Undergraduate Student  
Computer Science  
Aberystwyth University

October/November/December 2023

## Contents

1	C Program Overview .....	3
1.1	Program Structure and Design .....	4
1.2	Notable Design Features and Issues .....	5
1.2.1	Design Features.....	5
1.2.2	Issues.....	6
2.1	Testing and Robustness .....	7
2	My C++ Approach to the Program .....	8
2.1	Object-Oriented Design.....	9
2.1.1	Example:.....	9
2.2	Inheritance and Polymorphism.....	10
2.3	Exception Handling .....	11
2.4	Operator Overloading .....	11
2.4.1	Example:.....	11
2.5	Smart Pointers.....	12
2.5.1	Exclusive Ownership ('std::unique_ptr'): .....	12
2.5.2	Shared Ownership ('std::shared_ptr'):.....	12
2.5.3	Breaking Cycles ('std::weak_ptr'):.....	13
2.6	Standard Template Library .....	14
2.6.1	Example:.....	14
2.7	Conclusion.....	15

# 1 C Program Overview

This section of the report summarises the development of a Fractal Colouring Book Generator program, as outlined in the assignment brief<sup>1</sup>.

The program reads input from an NFSF (Neals Fractal Specification Format) file, interprets the specifications, and generates an SVG output file.

Accompanying this report is a screencast showing a clean building of the code, and the running of the code showing output from the provided NFSF input files.

---

<sup>1</sup> CS23820 Assignment (Online),  
[https://blackboard.aber.ac.uk/ultra/courses/\\_46444\\_1/outline/edit/document/\\_2658006\\_1?courseId=\\_46444\\_1&view=content](https://blackboard.aber.ac.uk/ultra/courses/_46444_1/outline/edit/document/_2658006_1?courseId=_46444_1&view=content) (2023)

Accessed: 30<sup>th</sup> October 2023

Note: Restricted Access (Aberystwyth University Blackboard)

## 1.1 Program Structure and Design

My implementation of the Fractal Colouring Book Generator was developed in a systematic and iterative manner. Firstly, I outlined the program's requirements. I designed the data structures for each instruction to store the required information. Throughout the development process and as I learned more about how the program should function, I continuously refined the code and tested the functionality using CLion's debugger and printing the information from each data structure to the console.

The program is divided into several files, each handling a different aspect of functionality. The main file ('main.c') first retrieves the input filename from the user, ensuring it has the '.txt' extension. It then parses the NFSF file using the 'nfsfParser' function, storing the parsed values in structures representing each instruction. The file then prompts the user to enter the output filename for the SVG file, appending '.svg' if required. Finally, the file generates a global transformation and then draws each shape of the fractal according to the parsed instructions. The SVG file is created with a canvas size of 700x700.

The NFSF parsing file ('file\_parser.c') extracts and stores the information from each instruction in the NFSF file into the corresponding structures located in the 'structures.h' header file. The code also includes error handling and validation for parsing, which ensures that the input file is correctly processed.

The fractal generator file ('fractal\_generator.c') uses the parsed information to recursively apply transforms to generate the fractal pattern. The generated output is written to the user specified SVG file. The code also includes error handling and validation for memory allocation.

## 1.2 Notable Design Features and Issues

### 1.2.1 Design Features

#### **Modularity and Code Organisation:**

- The program is divided into multiple files, each serving a specific functionality. This makes the code more organised and easier to maintain.

#### **NFSF Parsing:**

- The NFSF parsing file allows for the interpretation of Transform, Graphic, Branch, and Fractal instructions. This allows for flexibility as multiple instructions with different values can be extracted by the parser.

#### **Recursive Fractal Generation:**

- The program recursively generates the shapes based on the specified Transform, Graphic, Fractal and Branch data structures to create the fractal.
- Each shape is drawn in the SVG format, applying the rotations, translations, and scaling calculations defined in the code from Appendix B.3 and C in the assignment brief<sup>1</sup>.

#### **Error Handling:**

- The program includes different error-checking mechanisms, i.e., for file opening, parsing, etc to ensure the correct execution of functions. This enhances the programs robustness.

#### **Dynamic Memory Allocation:**

- The program handles memory dynamically, i.e., when translating and rotating points. This ensures efficient memory usage as there can be any number of coordinates.

#### **User Interaction:**

- The program prompts the user for an input and output filename, providing a user-friendly interface. If the user does not enter the file extension, then it is appended to the filename<sup>2</sup>.

---

<sup>2</sup> *strcat() in C (Online)*,  
<https://www.geeksforgeeks.org/strcat-in-c/> (2023)  
Accessed: 8<sup>th</sup> December 2023

## 1.2.2 Issues

### Segmentation Fault:

- Whilst modifying the 'file\_parser.c' file to handle multiple instructions of the same type, I encountered a segmentation error (Fig 1.1). After spending a considerable amount of time debugging the program, I found that the error was due to a stack overflow. I resolved this issue by setting a default value for every value in each data structure.<sup>3</sup>

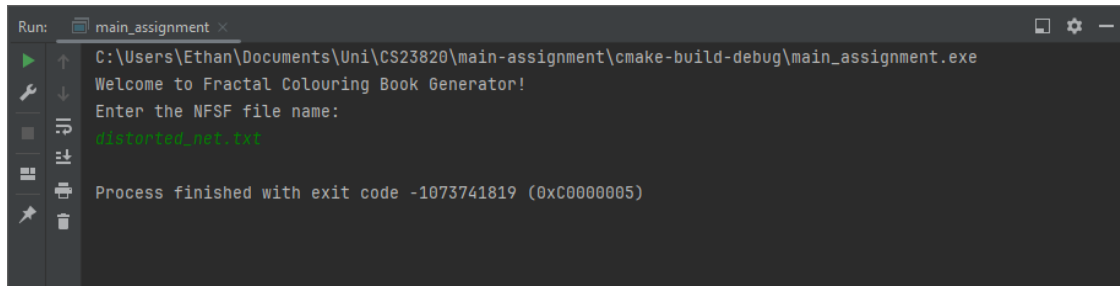


Fig 1.1: Segmentation fault when entering a file.

### Pointer Handling:

- Whilst extracting the values for each of the instructions, any value of type 'char' was not being correctly stored in the structures. This was because I was retrieving the address of the structure instead of accessing the value stored in the structure<sup>4</sup>.
- For example:*  
'&transform->name' did not correctly store the transform name to the structure.  
'transform->name' did correctly store the transform name to the structure.

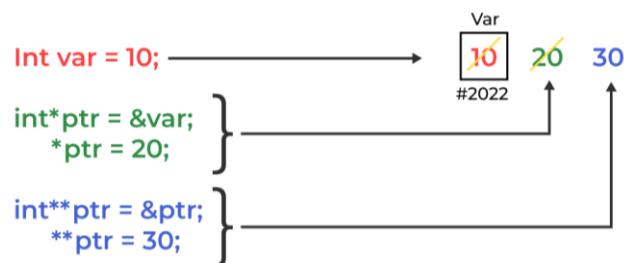


Fig 1.2: Diagram showing how pointers functions in C.<sup>5</sup>

<sup>3</sup> Segmentation Fault in C/C++ & How to Fix Them? (Online),  
<https://www.codingninjas.com/studio/library/what-is-the-segmentation-fault-in-cc> (2023)

Accessed: 21<sup>st</sup> November 2023

<sup>4</sup> Pointers in C: When to use the Ampersand and the Asterisk? (Online)  
<https://stackoverflow.com/questions/2094666/pointers-in-c-when-to-use-the-ampersand-and-the-asterisk#:~:text=%26%20takes%20a%20variable%20and%20gives,a%20pointer%20to%20a%20char%20>  
(2010)

Accessed: 10<sup>th</sup> November 2023

<sup>5</sup> Features and Use of Pointers in C/C++ (Online)  
<https://www.geeksforgeeks.org/features-and-use-of-pointers-in-c-c/> (2023)

Accessed: 10<sup>th</sup> November 2023

### Retrieving the Correct Number of Coordinates:

- As there could be any number of coordinates, I stored them in a two-dimensional array within the 'Graphic' structure. When I ran my program, the incorrect coordinates were being output. I resolved this by storing the number of coordinates in the 'Graphic' structure, then looping through the array storing the coordinates by this number.

### Interpreting Certain Branch Instructions:

- I incorrectly interpreted how certain fractal instructions should function; this included fractals with multiple branch instructions, or branch instructions leading to different fractals. I intended for my program to recursively create a tree with branch instructions.
- For example:

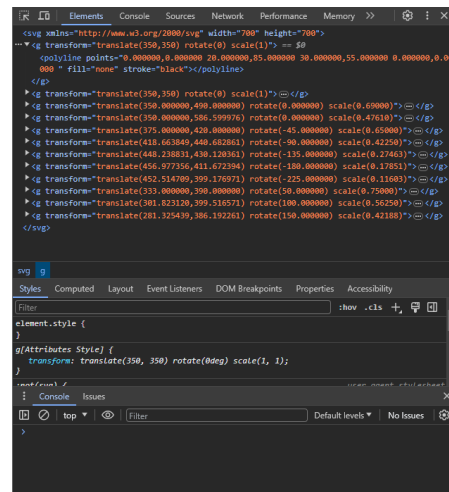
A screenshot of a web browser window displaying an SVG fractal tree. The browser's developer tools are open, showing the SVG code in the 'Elements' panel. The code includes a root element with a width of 700 and height of 700, followed by a series of nested 'g' (group) elements, each with a 'transform' attribute containing translation, rotation, and scale values. The 'Console' panel shows no errors. The browser's address bar shows the file path 'c:\s\...'. The fractal tree is rendered in the main content area, showing a complex branching structure.

Fig 1.3: Output of 'asymmetric\_branching\_tree.txt'.

## 2.1 Testing and Robustness

The program is robust, well-tested, and capable of generating a variety of fractal patterns based on the input file. It successfully handles the parsing of NFSF input files and outputting an SVG file with these values. Whilst the pattern generation method creates the expected output for certain NFSF files, it encounters errors for others.

- **Test Cases:**
  - Throughout the development, I tested the program with various NFSF files, covering different aspects of the specification.
  - For example, this included files with multiple fractals with multiple branches; graphic instructions with any number of coordinates; etc.
- **Error Handling:**
  - I ensured that the program contained robust error handling.
  - For example, this included ensuring that the program terminated without crashing; informative error messages; and making the program resilient to unexpected input.
- **User Input Handling:**
  - The program ensures that the user can only enter input files that exist in the program's directory.
  - If the user does not enter the file extension, then this is appended automatically.

## 2 My C++ Approach to the Program

This section of the report describes what features and design choices would be different from the C implementation of the Fractal Colouring Book Generator if it was developed in C++.



## 2.1 Object-Oriented Design

Compared to C, which is a procedural programming language, C++ allows for a more structured and object-oriented approach. This means that the properties and behaviour of each instruction type (Transform, Graphic, Fractal, Branch) could be encapsulated within its own class.

### 2.1.1 Example:

A 'Transform' class could have private member variables (e.g. 'name', 'rotation', etc.) and public member functions. This would improve modularity, code organisation, and encapsulation access specifiers could prevent direct access to internal details outside of the class.

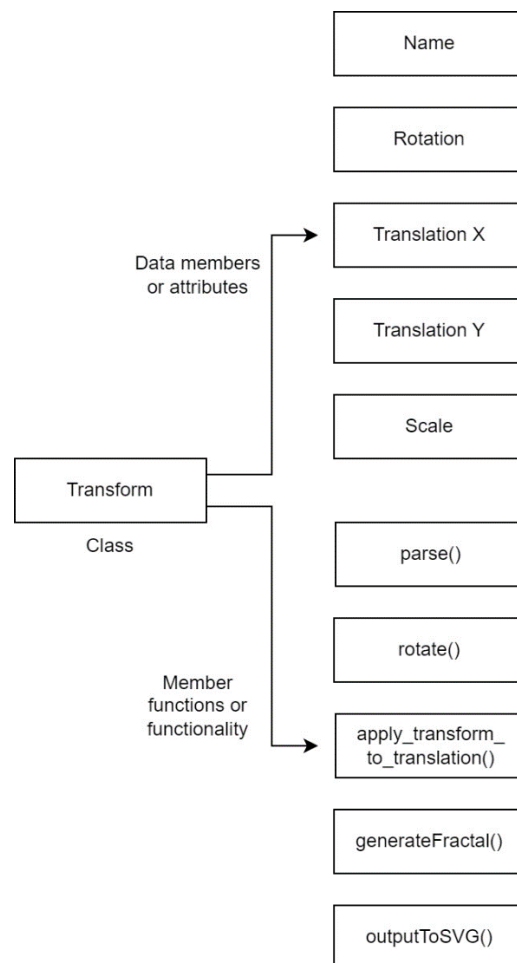


Fig 2.1: Diagram showing how the 'Transform' structure could be implemented in an object-oriented way.

## 2.2 Inheritance and Polymorphism

C++ supports inheritance which is used to establish hierarchical relationships between entities.

For example, a base class 'AllInstructions' could encapsulate shared properties and behaviours from different instruction types. This allows for a more organised and modular structure, as it allows for the addition of new entity types in the future without breaking existing functionality, reducing redundancy, and establishing clear relationships between entities (Fig 2.2).

Polymorphism allows for objects of derived classes to be treated as objects of their base class. This is useful for when there is a collection of entities of different types, as it allows for a single action to be performed in different ways.

For example, a base class 'AllInstructions' with the derived classes 'Transform' and 'Graphic' could both override the 'printInfo()' function to output the name (Fig 2.2).

```
class AllInstructions {
public:
    virtual void printInfo() const {
        std::cout << "Entity: " << name << std::endl;
    }

protected:
    char name[100]{};
};

class Transform : public AllInstructions {
public:
    void printInfo() const override {
        std::cout << "Transform: " << name << std::endl;
    }
};

class Graphic : public AllInstructions {
public:
    Graphic(const char string[3]) {

    }

    void printInfo() const override {
        std::cout << "Graphic: " << name << std::endl;
    }
};
```

Fig 2.2: C++ implementation of inheritance and polymorphism.

## 2.3 Exception Handling

C++ contains more advanced exception handling features; this would enhance the program's error-handling capabilities. Instead of relying on return codes, exceptions can be thrown, providing a cleaner, and more robust error-handling approach (Fig 2.3).

```
int main() {  
    try{  
        Transform t;  
        t.printInfo();  
    } catch (const std::exception& e) {  
        std::cerr << "An exception was thrown: " << e.what() << '\n';  
    }  
}
```

Fig 2.3: C++ implementation of error handling.

## 2.4 Operator Overloading

Operator overloading in C++ enhances code readability by making operations more intuitive and align closer with mathematical or logical expectations.<sup>6</sup>

### 2.4.1 Example:

I could use overloading operators such as '+' for the 'Translation' structure, '<<' for the 'Graphic' structure, and '==' for the 'Transform' structure (Fig 2.4).

```
int main() {  
    std::ostream& operator<<(std::ostream& os, const Graphic& graphic) {  
        os << "Graphic Name: " << graphic.name << ", Coordinates: ";  
        for (int i = 0; i < graphic.coordinateCount; ++i) {  
            os << "(" << graphic.coordinates[i][0] << ", " << graphic.coordinates[i][1] << ") ";  
        }  
        return os;  
    }  
}
```

Fig 2.4: C++ implementation of operator overloading for the Graphic structure.

---

<sup>6</sup> Operator Overloading in C++ (Online),  
<https://www.geeksforgeeks.org/operator-overloading-cpp/> (2023)  
Accessed: 1<sup>st</sup> December 2023

## 2.5 Smart Pointers

C++ uses smart pointers to enhance memory management, this provides a safer and more convenient alternative to C pointers as they offer automated memory management by automatically deallocating objects when they are no longer needed, making the code more robust and reliable. Therefore, I would not have to rely on manual allocation and deallocation using functions like 'malloc' and 'free'.

C++ supports three types of smart pointers: 'std::unique\_ptr', 'std::shared\_ptr', and 'std::weak\_ptr'.<sup>7</sup>

### 2.5.1 Exclusive Ownership ('std::unique\_ptr'):

- 'std::unique\_ptr' ensures that there is only one unique pointer to an object (Fig 2.5.1), which prevents multiple pointers from managing the same memory. When the 'std::unique\_ptr' goes out of scope, the memory is automatically deallocated, reducing the risk of memory leaks.

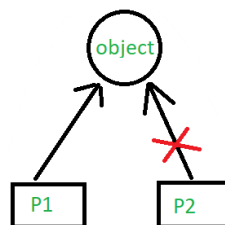


Fig 2.5.1: Diagram showing that 'std::unique\_ptr' stores only one pointer.

- For example, 'uniqueGraphic' could dynamically allocate and exclusively own a 'Graphic' object (Fig 2.5.2).

```
int main() {  
    std::unique_ptr<Graphic> uniqueGraphic = std::make_unique<Graphic>();  
}
```

Fig 2.5.2: C++ implementation of exclusive ownership.

### 2.5.2 Shared Ownership ('std::shared\_ptr'):

- 'std::shared\_ptr' allows for multiple pointers to share ownership of the same dynamically allocated object (Fig 2.6.1), this ensures that memory is deallocated only when the last 'std::shared\_ptr' pointing to the resource is destroyed by maintaining a reference count. This is useful when multiple parts of the program need to access the same object.

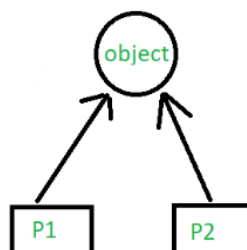


Fig 2.6.1: Diagram showing that 'std::shared\_ptr' can have more than one pointer.

---

<sup>7</sup> Smart Pointers in C++ (Online),  
<https://www.geeksforgeeks.org/smart-pointers-cpp/> (2023)  
Accessed: 2<sup>nd</sup> December 2023

- For example, 'sharedGraphic' could dynamically allocate and allow shared ownership of a 'Graphic' object, using reference counting to automatically manage memory (Fig 2.6.2).

```
int main() {  
    std::shared_ptr<Graphic> sharedGraphic = std::make_shared<Graphic>();  
}
```

Fig 2.6.2: C++ implementation of shared ownership.

### 2.5.3 Breaking Cycles ('std::weak\_ptr'):

- 'std::weak\_ptr' breaks ownership cycles that can lead to memory leaks. It allows for checking whether the resource it points to still exists without affecting its reference count.
- For example, 'weakGraphic' could be a non-owning reference for a 'Graphic', which prevents ownership cycles (Fig 2.7.1).

```
int main() {  
    std::unique_ptr<Graphic> uniqueGraphic = std::make_unique<Graphic>();  
    std::shared_ptr<Graphic> sharedGraphic = std::make_shared<Graphic>();  
  
    std::weak_ptr<Graphic> weakGraphic = sharedGraphic;  
}
```

Fig 2.7.1: C++ implementation of breaking cycles.

## 2.6 Standard Template Library

C++ contains a Standard Template Library which introduces useful tools for data management and manipulation. This provides a more efficient alternative to manual memory allocation. If I was to implement the Fractal Colouring Book Generator in C++, the following aspects of a Standard Template Library could be used:<sup>8</sup>

### 2.6.1 Example:

'std::vector' is a dynamic array that automatically handles resizing. This would simplify the storage of variable-sized data structures, such as the coordinates in the 'Graphic' structure (Fig 2.8).

'std::list' could be used for managing dynamic sequences of elements where insertions or removals are required, for example, on the varying number of coordinates in the 'Graphic' structure. This would allow for more efficient management of these sequences.

'std::map' could be used for associating values with keys, for example, associating the names of different instruction types with their corresponding objects.

```
int main() {
    std::vector<Graphic> graphicsVector;

    Graphic graphic1 = { "g1", {0,0}, {100,80}, {100,0}, {0,0}};
    Graphic graphic2 = { "g1", {0,0}, {100,80}, {100,0}, {0,0}};

    graphicsVector.push_back(graphic1);
    graphicsVector.push_back(graphic2);

    for (const Graphic& graphic : graphicsVector) {
        std::cout << "Graphic: " << graphic.name << std::endl;
    }

    return 0;
}
```

Fig 2.8: C++ implementation of a Standard Template Library.

---

<sup>8</sup> *The C++ Standard Template Library (Online)*,  
<https://www.geeksforgeeks.org/the-c-standard-template-library-stl/> (2023)  
Accessed: 2<sup>nd</sup> December 2023

## 2.7 Conclusion

In conclusion, if I implemented the Fractal Colouring Book Generator using C++, I would have significant enhancements in terms of modularity, robustness, and code organisation.

As C++ is an object-oriented language, it allows for a more structured design, so each instruction type could be encapsulated within its own class. Features such as inheritance, polymorphism, advanced exception handling, operator overloading, smart pointers, and the inclusion of the Standard Template Library allow for a more organisation, sophisticated, and efficient implementation compared to the original C implementation.

Overall, having looked at the assessment criteria, I believe I should be awarded a mark of 60% for this assignment.