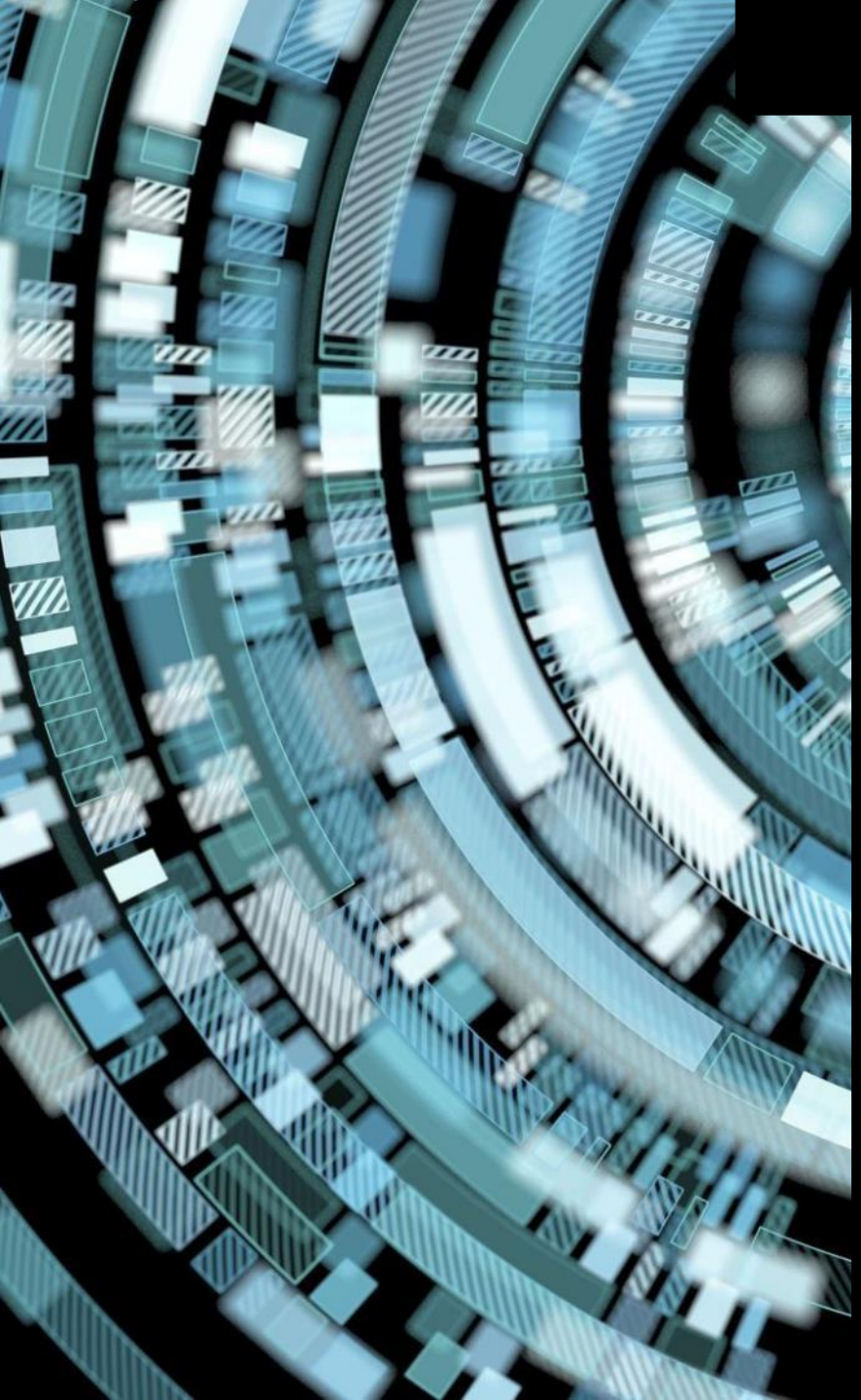
COM4013
Introduction to Software Development

- Week 11
- Basics of Programming in C++

- *Umar Arif – u.arif@leedstrinity.ac.uk*

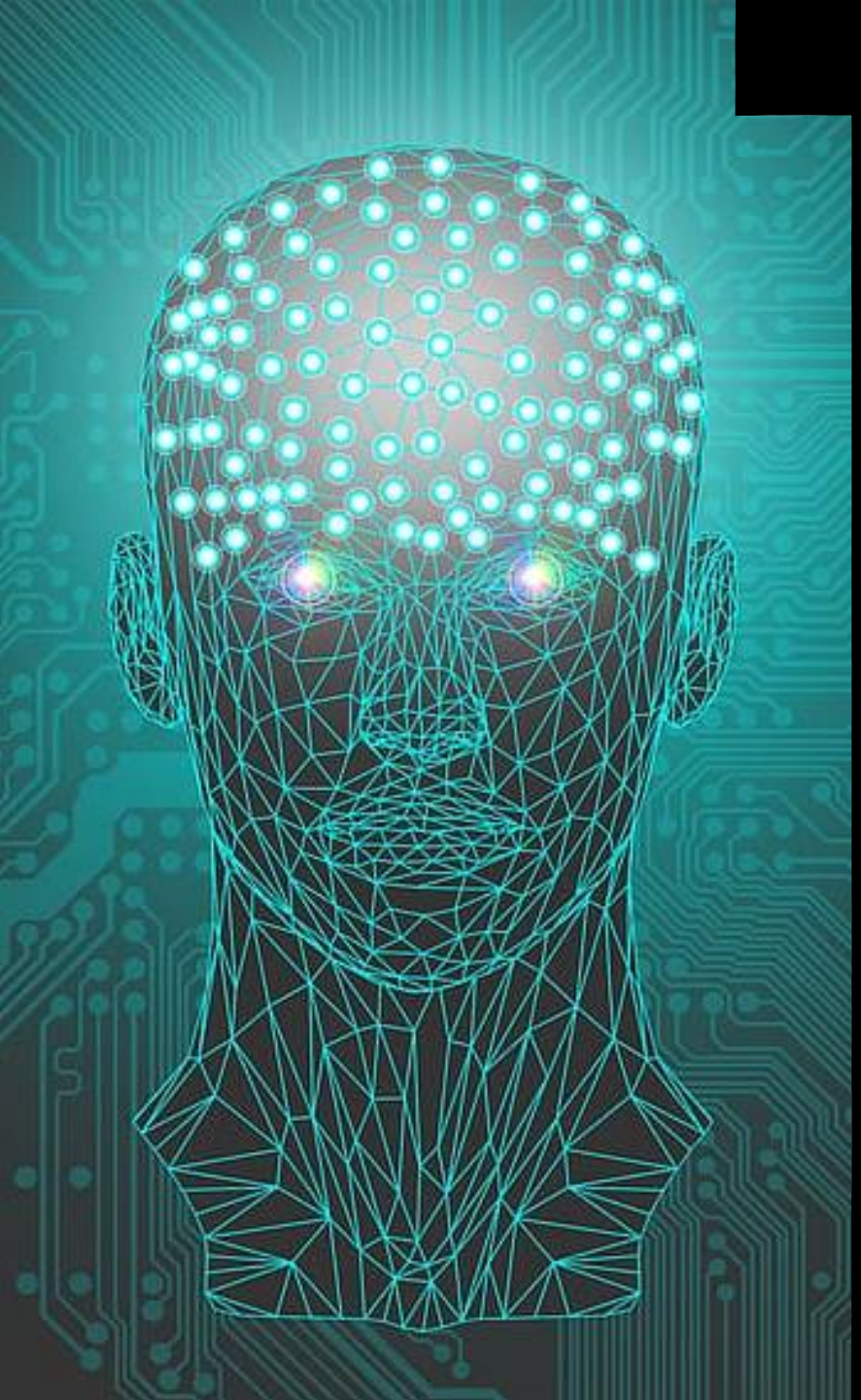


Reminder

- **Welcome back!**
- The deadline for **Assessment 1**, worth **70%** of the module's grade is midday on the 9th of February
- A turn-it-in box (for submissions) can be found on Moodle
- If you hand in the assessment late, the maximum mark you can attain for this assessment is 40%
- If you do not submit the assessment in a week after this point, then you will be given 0 marks

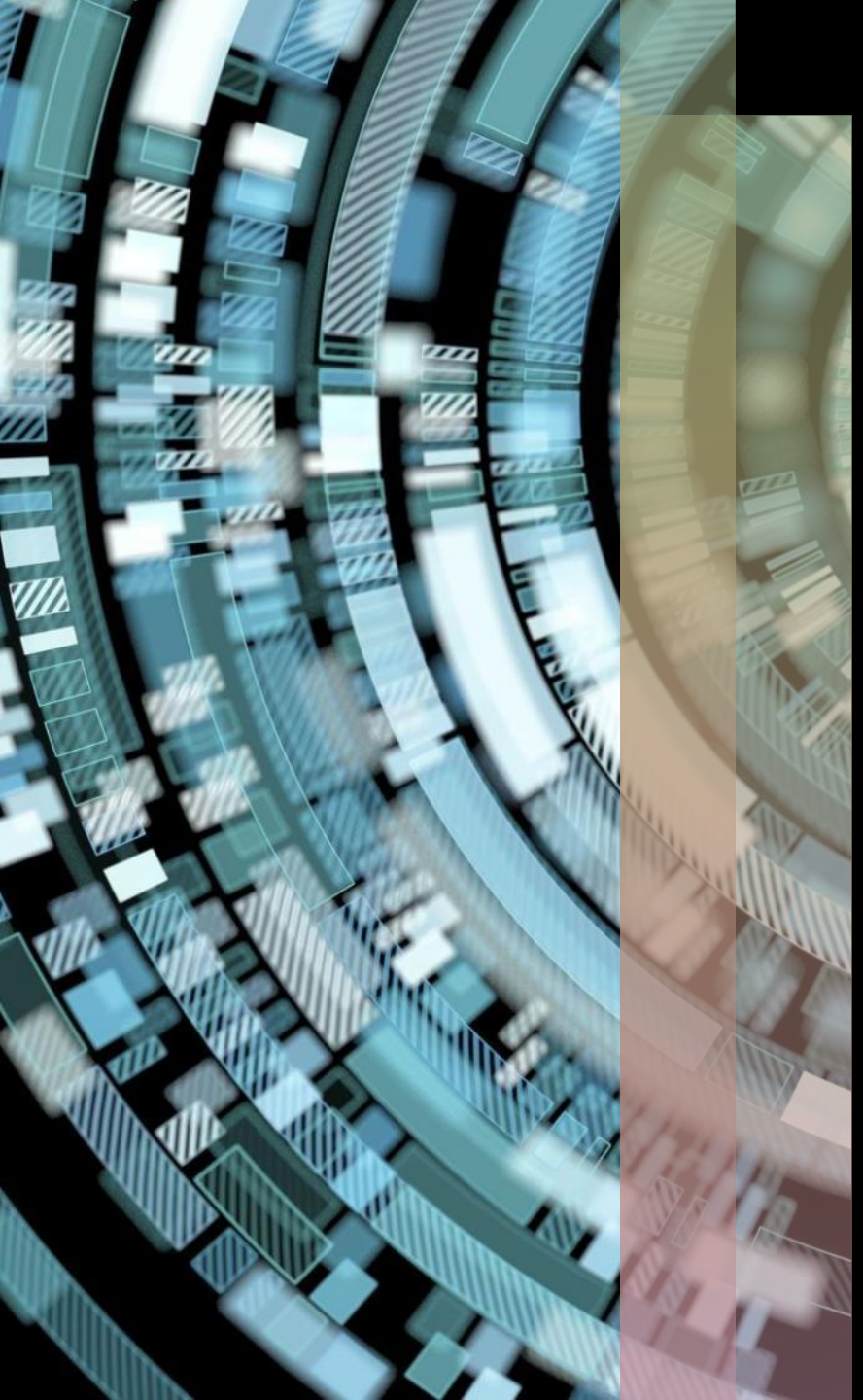
- In **Assessment 2** you will develop an electronic Python portfolio, constituting **30%** of the module grade, featuring refined versions of up to three **specified** exercises in the practical worksheets

- Any questions?



This Semester

- This semester we will be covering Algorithms and computational thinking using C++ as our main language
- As computer scientists we ubiquitously design and implement algorithms. They are fundamental, serving as the backbone of all computational processes
- Some of the Algorithms and C++ we will cover this semester:
  1. Introduction to C++ and C++ Block Statements
  2. Algorithms and Pseudocode and Functions
  3. Logic, Propositions, Truth Tables, and Structures
  4. Stack, Stack Class, and Arrays
  5. Recursion, Divide and Conquer, and Searching
  6. Search Algorithms, Constants, Global Variables, and Print
  7. Sort Algorithms and Strings
  8. Graphs, Graphs Class, Trees, Traversing, STL, and Vectors
  9. Algorithm Efficiency, Big-O Notation, and Iterators
  10. More... C++... And Summer Homework :)



Programming Languages

- A programming language is a **formal set of instructions** used to communicate with a computer. It provides a means for humans to instruct computers to **perform specific tasks**
- The choice of programming language significantly impacts the development process and the success of a project. Different languages excel in specific domains

- Popular languages and their domains:
  - C++ - Efficient, widely used in systems programming.
    - Unity uses C++ in its backend operations
  - Java - Platform-independent, commonly used in web development
  - Python – Known to be versatile, readable, easy to use, and the main language used for AI/Data Science related activities
  - JavaScript - Essential for web development, enables dynamic content (Node.js)

C++

- Python is an interpreted language whereas C++ is a fully compiled language
- Which means that it uses a compiler to convert C++ code into machine code
- Much of C++ is identical to Python (the syntax is slightly different and there's more rules)

```
for (int i = 0; i < 100; i++)
{
    cout << i << ", " << endl;
}
```

What are the key difference between this for loop and one written in Python?

A C++ Example

An example C++ program:

```
#include <iostream>
using namespace std;
// A simple hello world program
int main()
{
  // Output to screen
  cout << "Hello World!";
}
```

- All C++ programs must include the function main()
- Program execution begins with main()
- By convention, main() appears at the end of the source code
- Note that main() starts with a lower case ‘m’
- Let's say we wanted to also print out "My name is Umar", how could we go about doing this?

C++ Scope

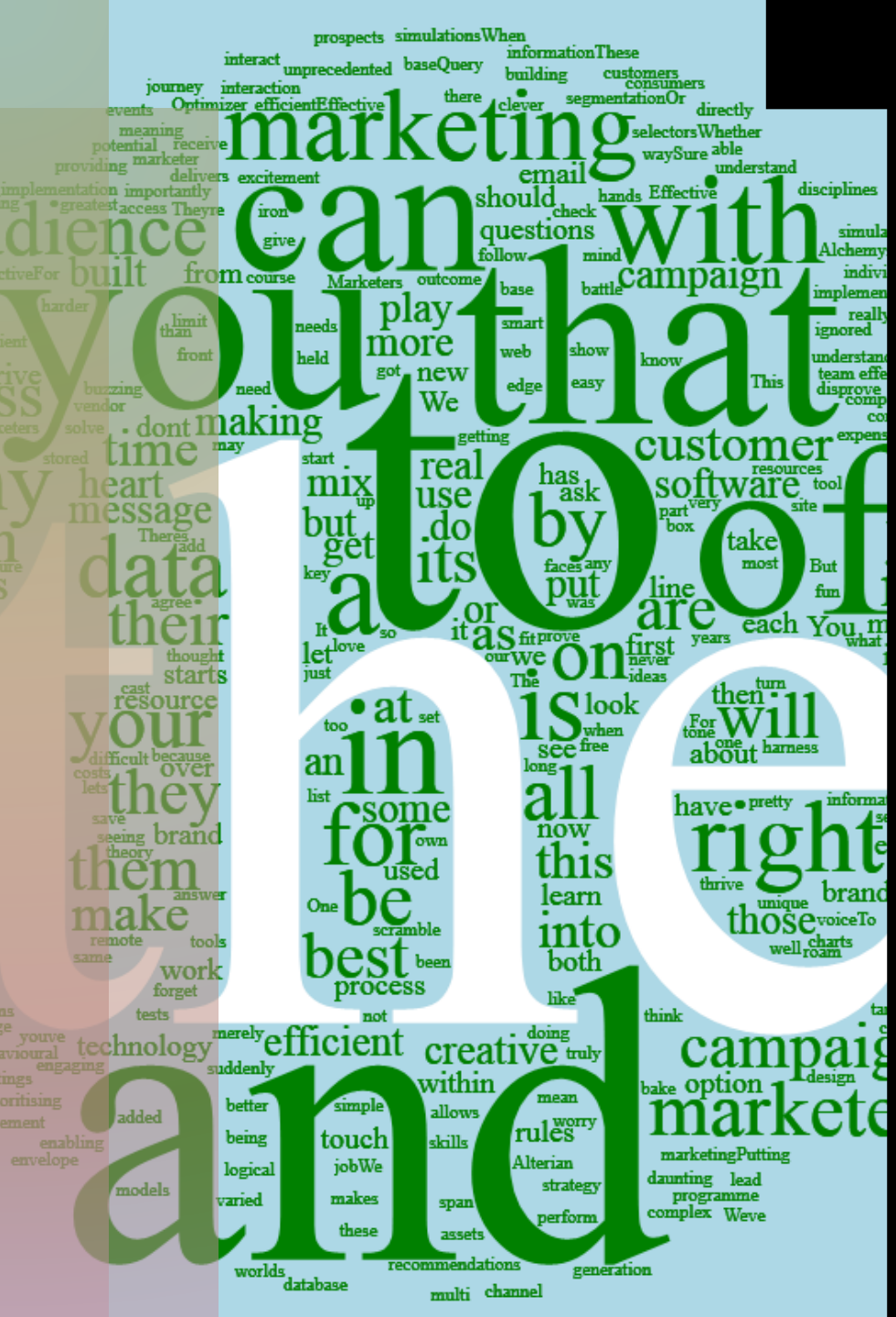
- Most of the functionality of C++ comes from the use of **libraries/dependencies**
- For example, to obtain the input and output commands, you need to **include** the **“iostream”** library
- C++ also uses **namespaces** which is a way of expressing **scope**
- For the moment just remember to include the namespace statement after including the libraries (as below)

```
#include <iostream>
using namespace std;
```

- Remember that C++ uses **block scope** NOT **function scope**. You must follow **good programming standard** to use C++ well. For instance, the main function (entryway into your code) is no longer optional
- Note that variable declarations no longer require **initialisation** in C++, **declaration** is a must (obviously)

C++ Variable Types

- Floating Point
  - A **decimal number**
  - Example: `float myNumber = 20.9f;`
- Double Point
  - A more **accurate decimal number**
  - Example: `double myNumber = 20.9;`
- Integer
  - Represents a **whole number**
  - Example: `int anotherNumber = 3;`
- Character
  - **Single character** enclosed in **single quotation marks**
  - Example: `char character = 'U';`
- Boolean
  - true or false value (unlike Python. lowercase in C++)
  - Example: `bool isAlive = true; // 0/1 works too`
- String (class)
  - **Multiple characters** enclosed in **double quotation marks**
  - Example: `string name = "Umar";`



Cin and Cout

- **cin** and **cout** are your input and output statements
- The direction of the angle brackets is important

```
#include <iostream>
using namespace std;
int main()
{
  int i;                          // intialised int
  cout << "enter a number";       // Console output
  cout << endl;                   // endl == "\n"
  cin >> i;                       // Console input
  cout << "The number you entered was " << i;
}
```

- **Iostream** is the library from which both cin and cout functions can be found
- Note that this library contains many more functions that makeup a part of the C++ **standard template library** (STL)

```
BFS(G, s)
1  for each vertex u ∈ V(G) \ {s}
2       color[u] = white
3       d[u] = ∞
4       π[u] = nil
5  color[s] = gray
6  d[s] = 0
7  π[s] = nil
8  Q = ∅
9  Enqueue(Q, s)
10 while Q ≠ ∅
11      u = Dequeue(Q)
12      for each v ∈ Adj[u]
13           if color[v] == white
14                color[v] = gray
15                d[v] = d[u] +
16                π[v] = u
17                Enqueue(Q, v)
18      color[u] = black
```

cin & cout

Cin is for input

```
int size;
cin >> size;
// into the variable
```

- Input is **directed into** a variable
- cin automatically tries to convert keyboard input into the data type of the variable
- cin will assume here that the data coming from the keyboard is an integer
- cin reads until it reaches a carriage return or a space.
- If you entered the phrase "once upon a time" at the keyboard, then it would only read in the word "once"
- You need another read to get the rest of the phrase

Validating Input

- C++ has no access to the parsing and checking routines. However, we can validate data types
- This check on the input stream will work with any data type

```
int i;
float f;
char ch;
cin >> i >> f >> ch;

if (cin.fail())
{
    cout << "an error has occurred";
}
```

- You will have to use your ingenuity to learn C++

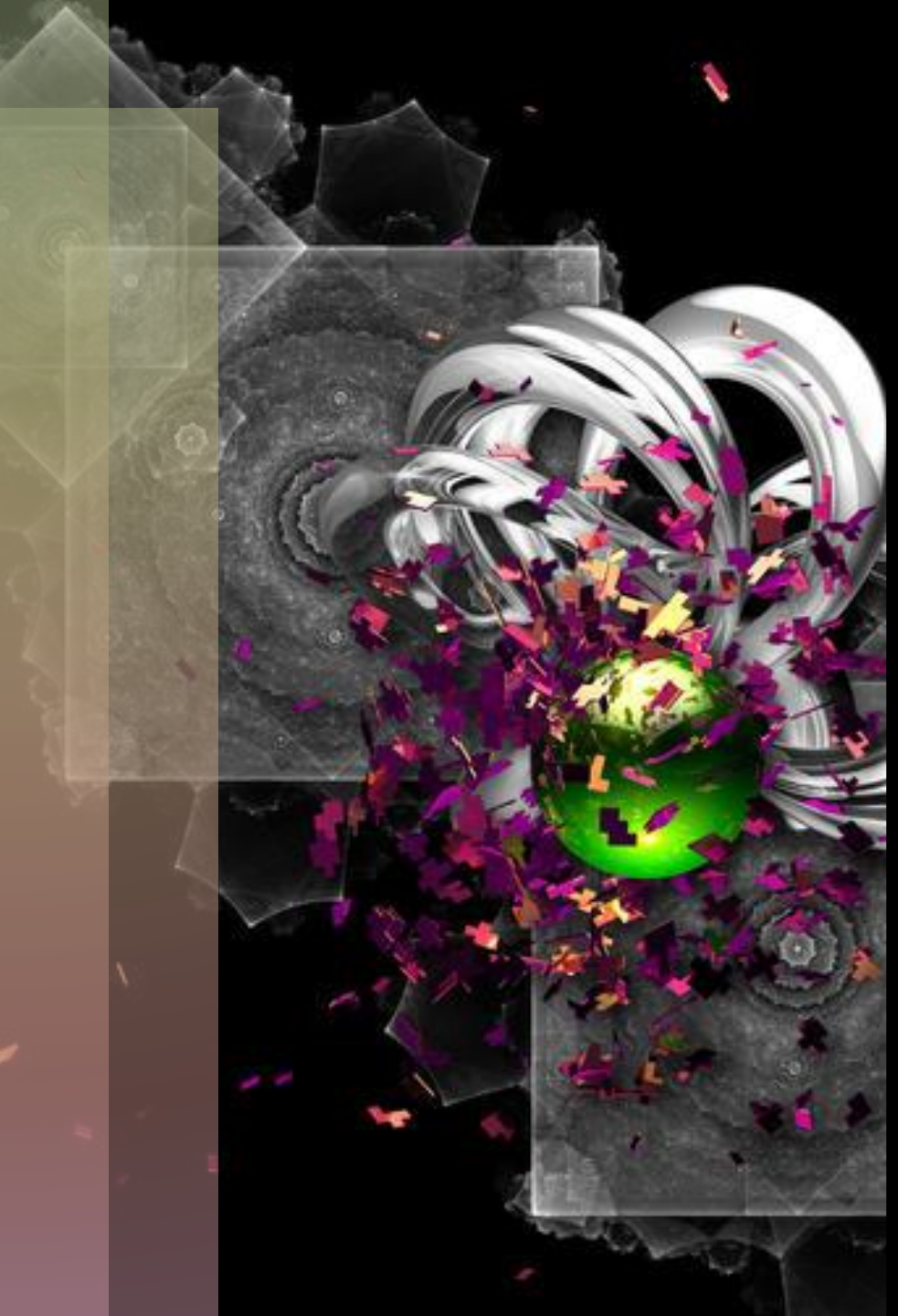
Flags

- It can also be useful to set up your flag and then use it to monitor all the different aspects of the input

```
bool error = false;
cin >> a >> b >> ch;
error = cin.fail();

If (ch != 'y')
{
  error = true;
}
```

Precedence

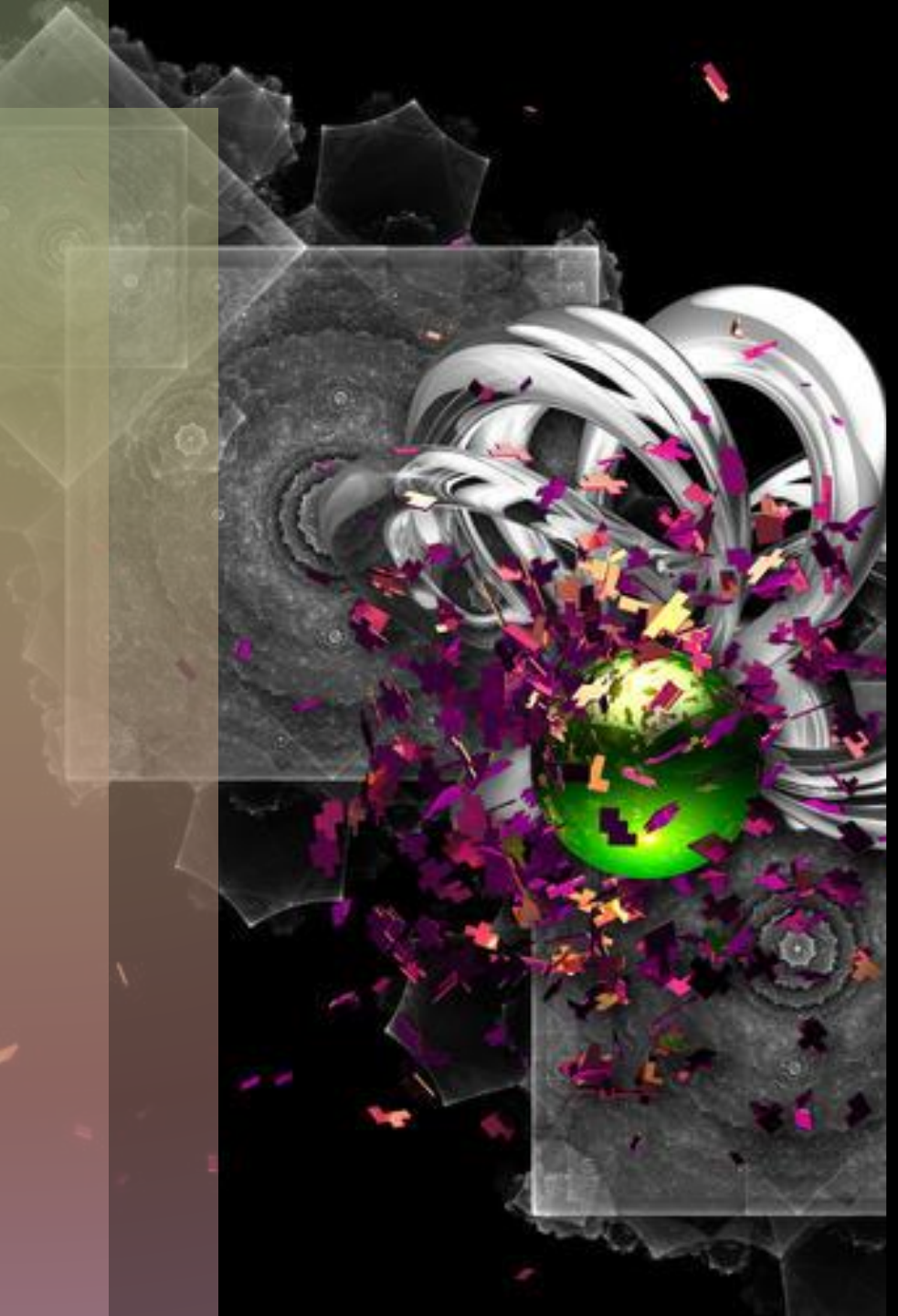
Precedence	Operator	Description
1	`::`	Scope resolution
2	`a++, a--`	Postfix increment, decrement
3	`()`	Function call
4	`. (dot), -> (arrow)`	Member access
5	`static_cast, dynamic_cast, const_cast, and reinterpret_cast`	Type conversion operators
6	`+, -, !, ~, ++a, --a, *ptr, &ptr, sizeof, typeid`	Unary operators
7	`*, /, %`	Multiplication, division, modulo
8	`+, -`	Addition, subtraction
9	`<<, >>`	Shift operators
10	`<, <=, >, >=`	Relational operators
11	`==, !=`	Equality operators
12	`&`	Bitwise AND
13	`^`	Bitwise XOR
14	`|`	Bitwise OR
15	`&&`	Logical AND
16	`||`	Logical OR
17	`? :`	Conditional (ternary) operator
18	`=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=`	Assignment operators
19	`,`	Comma operator



If Statements

```
#include <iostream>
using namespace std;

// Main function returns an int (default)
int main()
{
    int a = 10;

    if (a < 10)
    {
      cout << a << " is less than 10" << endl;
    }
    else if (a >= 10 && a <= 99) // elif equivalent
    {
      cout << a << " is >= 10 and <= 99" << "\n";
    }
    else
    {
      cout << a << " is greater than 99" << endl;
    }
}
```

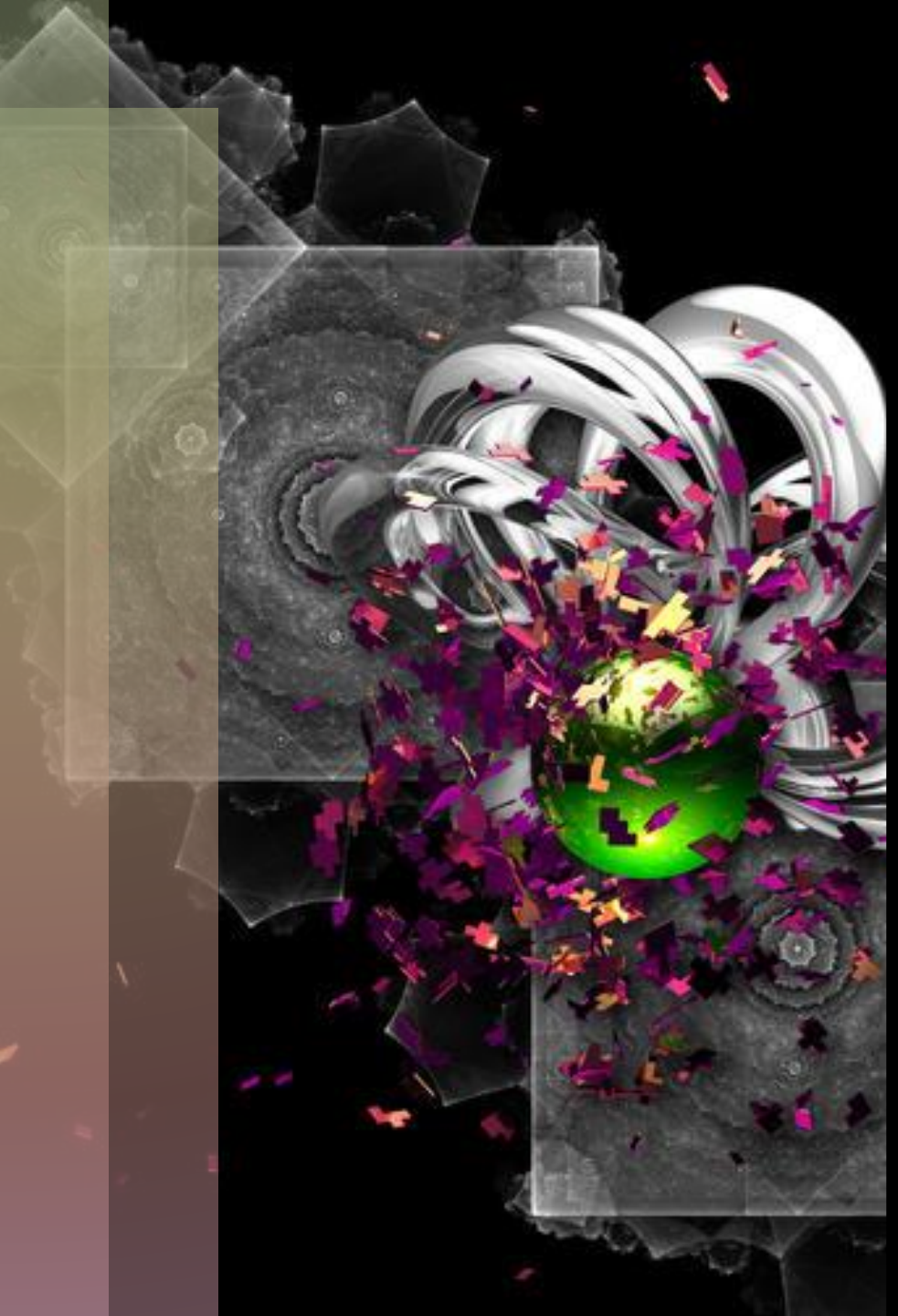
What is printed out here?



For Loops

```cpp
#include <iostream>
using namespace std;

int main()
{
    // For loop - quivalent of for in range
    // Could use unsigned int (positive values only)
    for (int i=0; i < 10; i++)
    {
      cout << i << endl;
    }

    // Array declaration
    int array1[3] = { 3, 4, 6 };

    for (int element : array1)
    {
      cout << element << endl;
    }
}
```

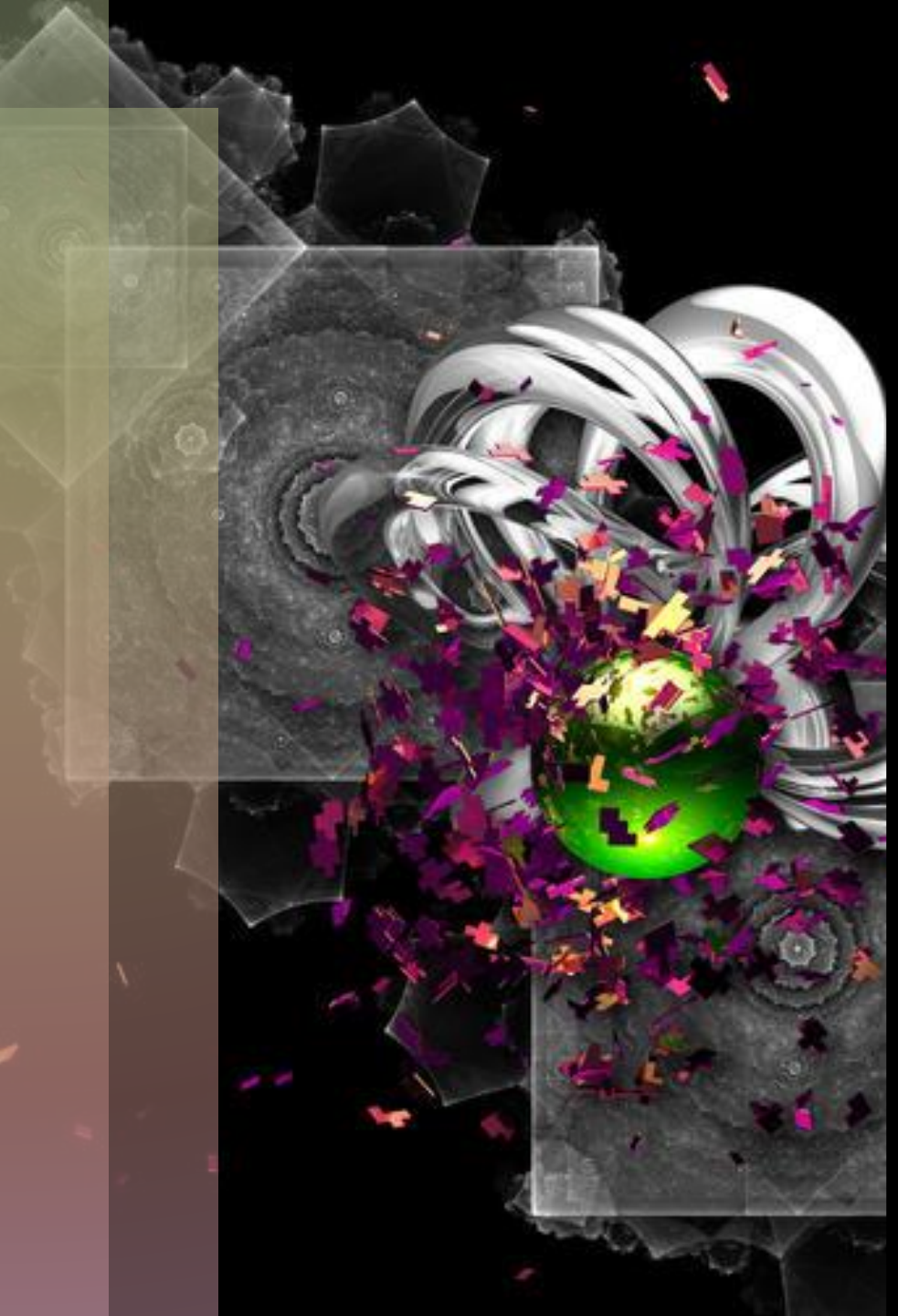
What is the difference between these for loops?



While Loops

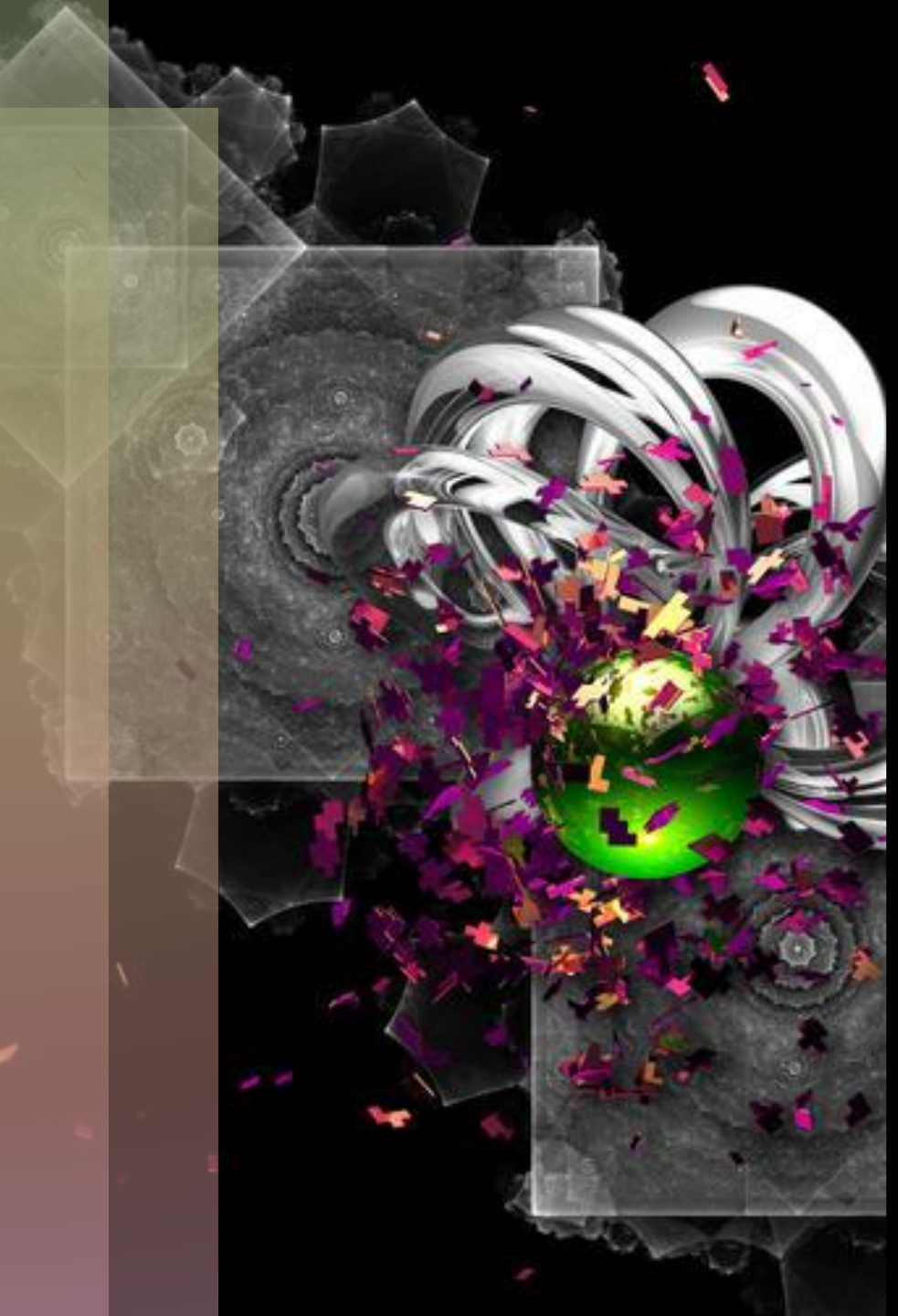
```cpp
#include <iostream>
using namespace std;

int main()
{
    int count = 0;
    while (count < 5) // Same as before
    {
      cout << "Count: " << count << endl;
      count++;
    }

    count = 0; // Reset the counter or this won't work
    do
    {
      cout << "Count: " << count << endl;
      count++;
    } while (count < 5)
}
```

What is the difference between these while loops?



Functions

```
#include <iostream>
using namespace std;
// Functions must be declared before the main method
// Returns a string and takes a string parameter
string HelloName(string name)
{
    return "Hello " + name;
}

// Returns an int and takes a string parameter
int StrLength(string n)
{
     return n.length() // string library method
}

int main()
{
    string name = "Umar";
    cout << HelloName(name) << ":" << StrLength(name)
    << endl;
}
```

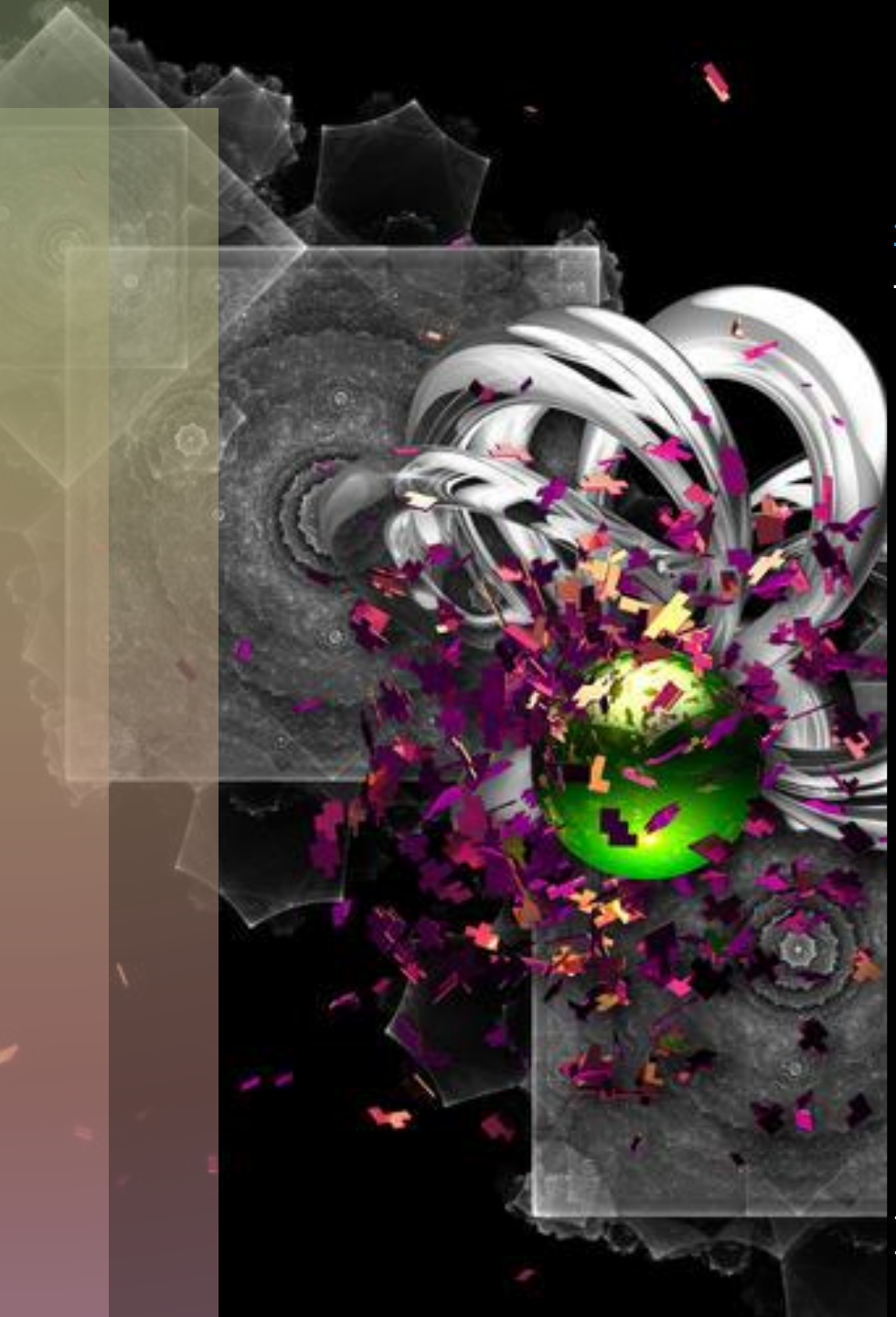


Arrays

```
#include <iostream>
using namespace std;

int main()
{
    int a2[5];

    for (int i=0; i < sizeof(a2) / sizeof(a2[0]); i++)
    {
        a2[i] = (i * 2);
    }

    for (int i : a2)
    {
        cout << i << endl;
    }

}
```

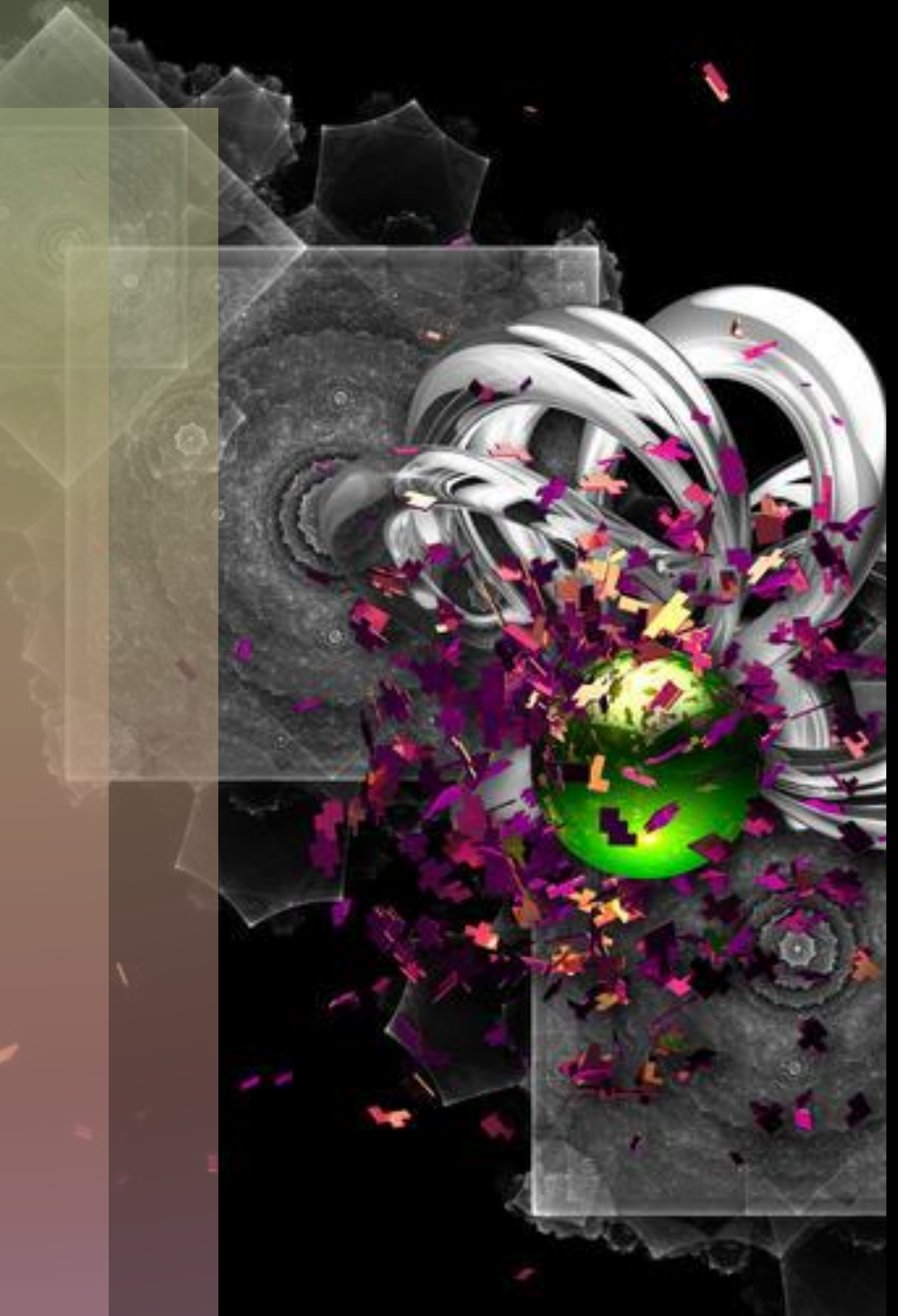
What is printed out here?

Input Validation



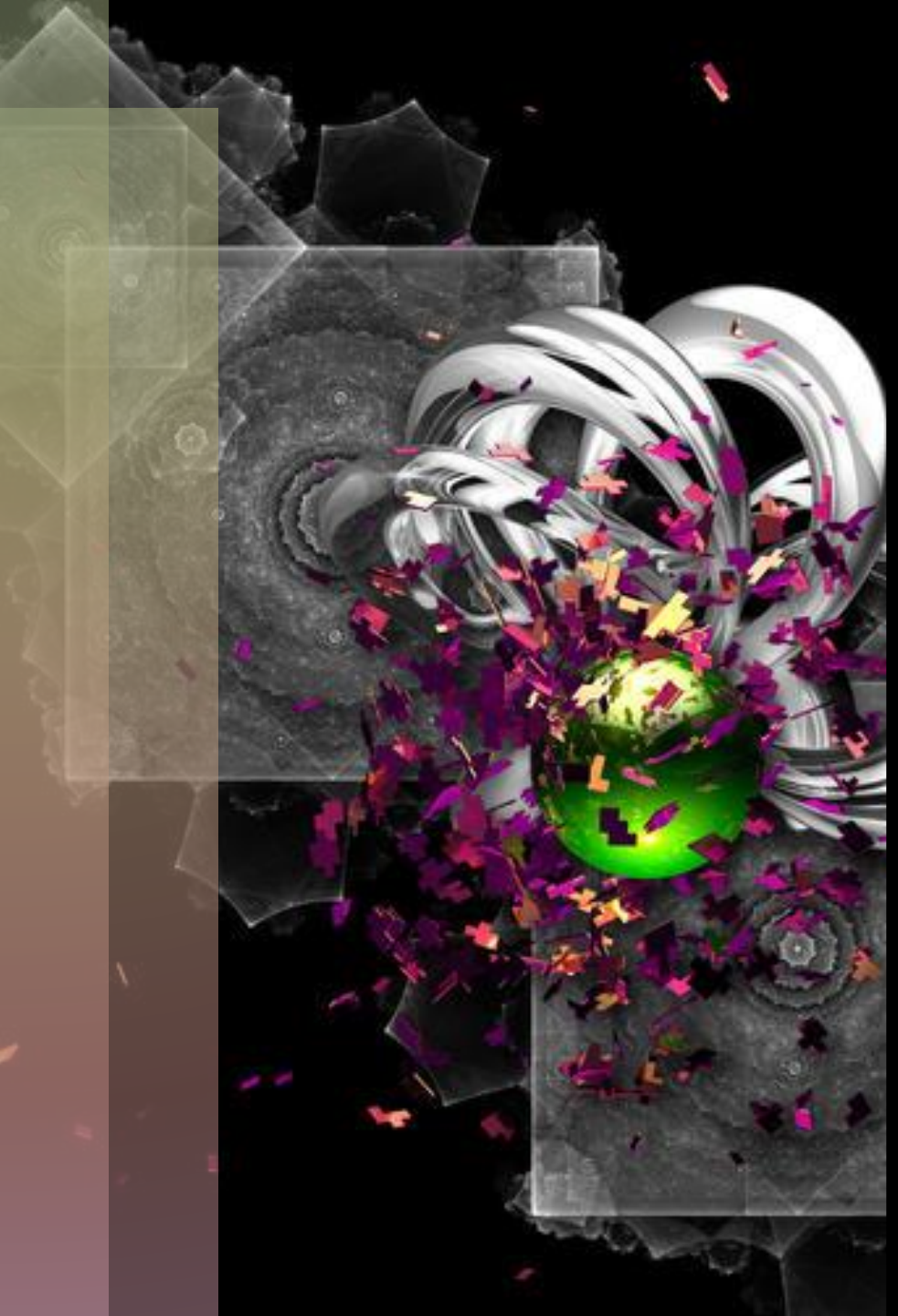
```cpp
int main()
{
    int i;

    while (true)
    {
        cout << "Enter a number: ";
        if (cin >> i)
        {
            // Valid input
            break;
        }
        else
        {
            cerr << "Invalid input" << endl;
            cin.clear(); // Clear the error flag
            // Discard invalid input
            cin.ignore(numeric_limits<streamsize>::max(), '\n');
        }
    }
}
```

Much more difficult



Structures

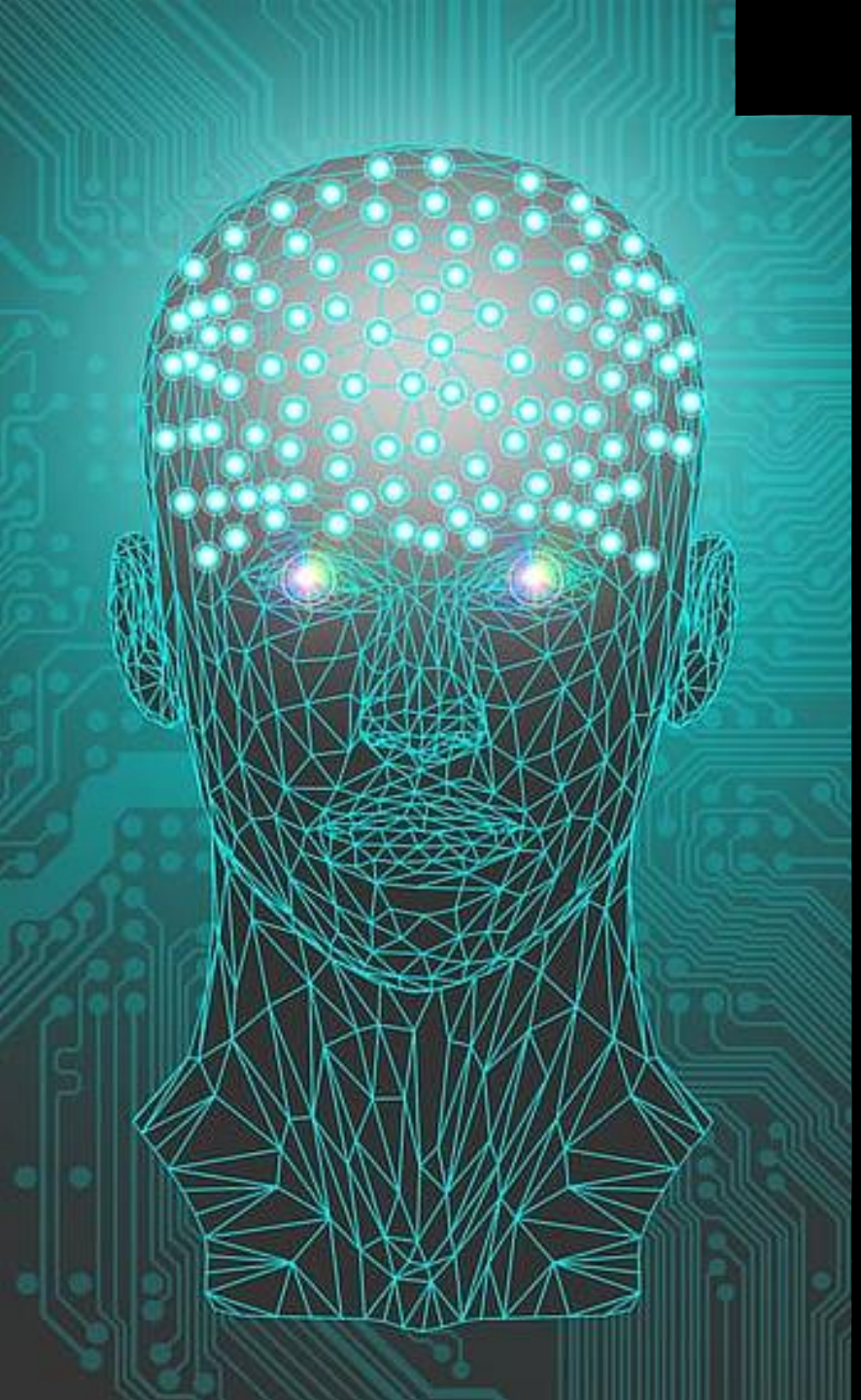
```cpp
#include <iostream>
using namespace std;

// Like classes but with public access and no OOPness
struct SStudent
{
    string name;
    int mark;
};

int main()
{
    SStudent s1 = { "Umar", 100 };
    cout << s1.name << " : " << s1.mark << endl;
}
```



Classes

```cpp
#include <iostream>
using namespace std;

class Cstudent // default private so specify
{
public:
    string mName;
    int mMark;

    CStudent(string name, int mark)
    {
        SetName(name); // will set name to mName
        SetMark(mark); // Need to write the set method
    }
};

int main()
{
    CStudent s1( "Umar", 100 );
    cout << s1.mName << " : " << s1.mMark << endl;
}
```

We also have ~CStudent() {} destructor



Scope

- C++ has "scope" rules.
- **Scope** means the area of the code in which the variable can be used
- It is the area of the program to which the variable is **visible**
- A variable name can be repeated if the scope of the two names is **different**
- Essentially the scope of a variable is the block of code in which it's declared
  - You could define this as the area under a block specified by curly braces
  - You define something in main, you can use it anywhere within main – and not in functions specified outside or vice-versa
- For a function, the scope of a variable is the function in which it is declared
  - A variable declared in one function is not visible to other functions

Scope

LOCAL VARIABLES

```
#include <iostream>
using namespace std;

int main()
{
    // local to main()
    int counter1;

    // body of main
}
```

LOCAL TO A FUNCTION

```
void foo(int number)  // number local to foo()
{
  cout << number;
}

int main()
{
  int value = 4;
  foo( value )
}
```

Area of a Square

```
float CalcArea()
{
  float width = 5.7;
  float height = 4.3;
  float area;
  area = width * height;

  return area;
}

int main()
{
  cout << "box area is ";
  float area = CalcArea();
  cout << area << "\n";
}
```

- The function on the left is of **limited** use because it is too **specific**
- Better to have a general-purpose function (below)

```
//Calculate the area of a square
float CalcArea(float width, float height)
{
  float area;
  area = width * height;
  return area;
}

int main()
{
  float area = CalcArea( 32.5, 17.0 );
  cout << area << endl;
}
```

Parameters and Return Types Recap

Parameters

- The function is flexible because data can be passed to it
- The parameters written in the function definition are the formal parameters
- The parameters supplied used when the function is invoked are the actual parameters
- The word arguments is used by some instead of parameters

"return"

- A function can send data back using the return statement
- Specifies the value to be returned by the call of the function
- Causes a dynamic exit from the function
- Need a variable to catch the data returned
- Use the value returned in a statement
- Ignore what has been returned

Parameters and Return Types Recap

Parameters Types

- Two types of parameters
  - call by copy
  - call by reference

- When a function is called, the value of the parameters are calculated and passed to the function for execution
- Within the function, the parameters act like local variables
- The value of the variables used in the parameter list are unchanged

- Termed "call by copy" because a copy of the variable is made
- Written exactly as the parameter list and function invocation have been done up until now

```
void CallByCopy( int number )
{
  number = 30; // number is changed,
}              // but this is local

int main()
{
  int a = 10;       // 10 assigned to ‘a’
  CallByCopy( a );
  cout << a;        // a still remains 10
}
```

Parameters and Return Types Recap

Parameters Types

- **Call by reference** using the **reference parameter** method is done simply by placing the ampersand symbol in front of the name of the variable

```
void FunctionName( int& value )
```

- Subsequent use of the variable is per normal
- You can mix call by copy and call by reference in you function definitions

- Call by reference is faster than call by copy
- The return statement only allows one data type to be sent back from a function
- Call by reference allows several data types to be sent back from a function
- (Return allows the use of dynamic exit, use of a function within a condition, etc. and this can be useful and lead to more elegant code)

```
void ReferanceParameter( int& number )
{
  number = 30; // original is changed,
}

int main()
{
  int a = 10;  // 10 assigned to ‘a’
  ReferanceParameter( a );
  cout << a;                  // a now 30
}
```

Summary

We've covered quite a few topics today:

- Introducing C++
- Block Scope
- Call by Copy/Reference

Any questions?

Resources

- C++ Tutorial (w3schools.com)
- cplusplus.com/doc/
- https://www.linkedin.com/learning/search?keywords=c%2B%2B%20syntax&u=0