Program Structure

The program runs line by line, from top to bottom. The first line instructs the compiler to locate the file that contains a library called iostream. This library contains code that allows for input and output. The main() function houses all the instructions for the program and in C++ has an int return. In Java, the equivalent is the main method: public static void main(String args[]). When a program is run, the compiler looks for the main() function as its entry point, regardless of language.

```
1  #include <iostream>
2
3  int main()
4  {
5      //Stuff
6  }
```

Printing to Console/Terminal

The character output stream is std::cout, equivalent to System.out.println() in Java. It is followed by the symbols <<, which is also between each value to be displayed.

```
3  int main()
4  {
5      std::cout << "Hello World, " << "Java > C++";
6  }
```

New Line and New Tab

The escape sequence \n generates a new line in a text string. The sequence \t generates a tab in a text sequence. This is exactly like Java.

```
3  int main()
4  {
5      std::cout << "Hello World\n";
6      std::cout << "\tKashmir Zindabad";
7  }
```

Compile and Execute

Unlike Java, C++ is a compiled language, meaning simply pressing "run" will not execute your code. You must first compile, then execute Using GNU, the compilation command is g++ followed by the file name. Here, the name of the source file is hello.cpp. Afterwards, you can either execute with the command ./a.out or if you add the command –o example, you can type ./example.

```
g++ hello.cpp
./a.out
```

```
g++ hellp.cpp -o hello
./hello
```

Comments

Comments in C++ are exactly the same as Java. For a single line comment, use // and for a multiline comment, use /* */. The compiler will always ignore comments. The purpose of comments is to improve readability of code for other programmers to examine.

```
3  int main()
4  {
5      //This is a single line comment
6
7      /*
8      This is
9      a multi
10     line comment
11     */
12 }
```

Variable Types

Type	Used for:	Range
short	small positive or negative integers	–32,768 to 32,767
unsigned short	only positive small integers	0 to 65,535
int	big positive or negative integer	–2,147,483,648 to 2,147,483,647
unsigned int	only positive big integers	0 to 4,294,967,295
float	positive or negative number with some decimals	bruh dw about it
double	positive or negative number with a lot of decimals	bruh dw about it
bool	true or false	true, false
char	Character	–
std::string	words	–

To declare a variable, the following structure is followed: type variable_name = value;. Note that a string is explicitly an object in C++ (whereas in Java it is a hybrid-object), hence the std:: prefix. C++ also supports unsigned variables

Taking Input from the Console

The character input stream is std::cin, equivalent to the Scanner() class in Java. It reads user input from the keyboard in the console.

```
3  int main()
4  {
5      int age = 0;
6
7      std::cin >> age;
8  }
```

Arithmetic Operators

C++ supports different types of arithmetic operators that can perform common mathematical operations. This includes addition, subtraction, multiplication, division, and finding the remainder. It is exactly the same as it is in Java.

```
4  int main()
5  {
6      int num1 = 2;
7      int num2 = num1 + 5; //num2 is 7
8      int num3 = num1 - num2; //num3 is -3
9      int num4 = num1 * num2; //num4 is 14
10     int num5 = (3 * num4) / 2 ; //num5 is 21
11     int num6 = num2 % num1; //num 6 is 1
12 }
```

Conditionals (If/Else)

An if statement is used to test an expression for truth. If the condition evaluates to true, then the code within the block is executed; otherwise, it will be skipped. An else if is the next statement checked. At the end, the else statement is run if none of the other conditions are true. It's the same functionality and syntax in Java as is in C++.

```
9      if (age < 16) {
10         std::cout<<"You can't drive";
11     }
12     else if (age < 18) {
13         std::cout<<"You're not an adult";
14     }
15     else {
16         std::cout<<"You're an adult";
```

Switch Statements

A switch statement provides a means of checking an expression against various cases. If there is a match, the code within starts to execute. The break keyword can be used to terminate a case. The default case is executed when no case matches. Switch statement's function and have the exact same syntax in Java, as they do in C++.

```
switch (grade) {
    case 9:
        std::cout<<"Freshman";
        break;
    case 10:
        std::cout<<"Sophmore";
        break;
    case 11:
        std::cout<<"Junior";
        break;
    case 12:
        std::cout<<"Senior";
        break;
    default:
        std::cout<<"Error";
    break;
```

Relational and Logical Operators

Operator	Meaning
==	equals
!=	not equal
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
&&	AND
||	OR
!	NOT

Loops

A while loop statement repeatedly executes the code block within as long as the condition is true. A for loop executes a code block a specific number of times. Loops have the same syntax and functionality as in Java. Loops can have break; statements to prematurely end the loop or continue; to skip over one iteration of the loop. If continue is executed, the current loop iteration will immediately end, and the next iteration will begin.

```
while (is_game_running)
{
    //code
}

for (int i = 0; i < target; i++)
{
    //code
}
```

Vectors and Index

A dynamic list, used to store a group of variables of the same type. To use it, #include the vector library at the top of the program. It is like an ArrayList() in Java. The index refers to the position in a vector. The first index is 0 and the last index is the size minus 1. The .size() function returns the size of the vector, .push_back(element) adds an element to the back of the list, and .pop_back() removes the last element.

```
#include <iostream>
#include <vector>
int main()
    {
        std::vector<double> list = {1.2, 3.4, -9.4}
        double first_element = list[0];
        double last_element = list[2];
        int size_of_list = list.size(); //Will return 3
        list.push_back(-2.46); //Adds -2.46 to list
        list.pop_back(); //Removes it from list
    }
```

Functions

Abtraction is the object-oriented principle of splitting code into segments. In C++, this is done through functions, equivalent to methods in Java. Functions can return something or have no return type (void), and they can have parameters to take.

```cpp
void print_message() {
    std::cout<<"Bruh";
}

int square(int num) {
    return num*num;
}

int main() {
  print_message();
  int i = square(2);
}
```

Function Declaration & Definition

A functions declaration can be done in a header file (.hpp or .h), and the definitions (body of the function) can be put in a separate .cpp file. The linker in a C++ complier will automatically sort it out for you.

```cpp
// ~~~~~~ main.cpp ~~~~~~
#include <iostream>
#include "functions.hpp"

int main() {
  std::cout << say_hi("Sabaa");
}
```

```cpp
// ~~~~~~ functions.hpp ~~~~~~

// function declaration
std::string say_hi(std::string name);
```

```cpp
// ~~~~~~ functions.cpp ~~~~~~
#include <string>
#include "functions.hpp"

// function defintion
std::string say_hi(std::string name) {
  return "Hey there, " + name + "!\n";
}
```

Templates and Function-Overloading

A function template is a C++ tool that allows programmers to add data types as parameters, enabling a function to behave the same with different types of parameters (called function-overloading). In the example, a template is created for the general arbitrary type T. When the main method passes it int parameters or double parameters, it will work either way.

```cpp
template <typename T>
T myDiff (T x, T y)
{
    return x - y;
}
```

Classes and Objects

An object (multiple) is essentially a variable you can create. A class (one) is a template for an object Attributes consist of info (usually variables) about an instance of the class and are private by default (other classes cannot access). A class can also have its own functions, which can be public or private. A constructor is a special kind of function that is called upon creation of a new object. Unlike Java, C++ has no garbage collector, instead the programmer has to manually take care of killing objects through a destructor.

```cpp
#include "city.hpp"
class City
{
  std::string name;
  int population;

public:
  City::City(std::string new_name, int new_pop) { //Constructor
      name = new_name;
      population = new_pop;
  }
  City::~City() { //Destructor
      //final clean-up
  }
private:
    City::bool is_capital() {
    }
```

```cpp
int main()
{
```

References and Memory Addresses

In C++, a reference variable is an alias for another variable or object. It is created using the & sign. Anything done to the reference also happens to the original. Aliases cannot be changed to an alias of something else. To obtain the memory address, also use the & sign (different from references). When & is used in a declaration, it is a reference operator. When & is not used in a declaration, it is memory address operator. In a function, references can be used to modify the value of the arguments, while avoiding a return statement.

```
1  int main()
2  {
3     int num_of_soda = 99;
4     int &num_of_pop = num_of_soda; //reference
5
6     std::cout << &num_of_soda; //Will return memory adress
7
8  }
```

Pass-by-Value vs Pass-by-Reference

When parameters were passed to a function, normal variables were used and then returned to the main method, which is called pass-by-value. However, by using reference variables, the value of the arguments can be modified while avoiding a return statement.

```
1  int main() {
2     int a = 100;
3     int b = 200
4     swap_num(a, b);
5  }
6
7  void swap_num(int &num1, int &num2)
8  {
9        int temp = num1;
10       int num1 = num2;
11       int num2 = temp;
12 }
```

Pointers and Dereferences

A pointer variable *points* to the memory address of something else, which is expressed in hexadecimal. It is created using the * sign. It is like references, but it is a relic of C and should be avoided. To obtain the memory address in a dereference also use the * sign (different from references). When * is used in a declaration, it is creating a pointer. When * is not used in a declaration, it is a dereference operator.

```
1  int main() {
2     int nice = 69;
3     int* ptr = &nice; //ptr points to the memory of nice
4     std::cout << ptr; //Will print the memory adress of nice
5     std::cout << *ptr; //Dereference prints nice itself
6  }
```