ELYSIUM SPARK NOTES

ELYSIUM ACADEMY
Milestone of Cognizance

C++ PROGRAMMING

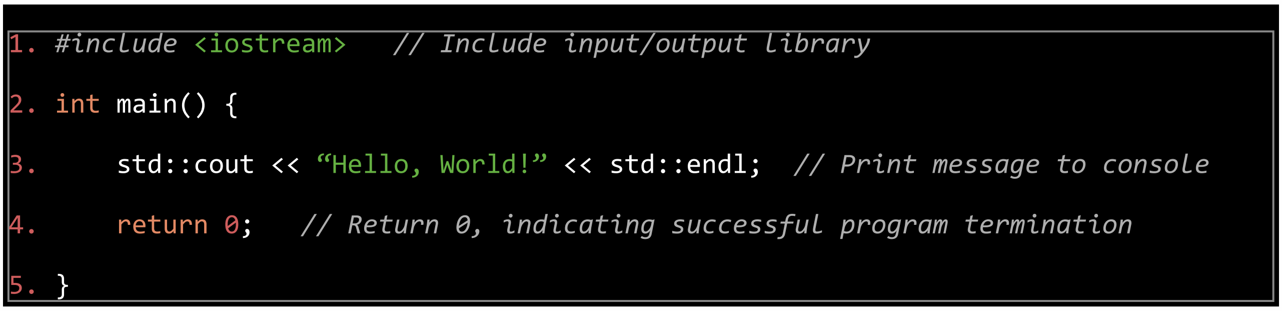
Elysium Academy Spark Notes

VERSION 2.4

01. C++ Basics

C++ is a compiled language, meaning you write source code, compile it to machine code using a compiler (like GCC or Clang), and then execute the compiled program.

Basic C++ Program



```cpp
#include <iostream>   // Include input/output library
int main() {
    std::cout << “Hello, World!” << std::endl;  // Print message to console
    return 0;   // Return 0, indicating successful program termination
}
```

- **Key Points:**
  - #include <iostream>: This includes the Input/Output stream library, which allows us to use std::cout and std::cin.
  - main() Function: The entry point for any C++ program. The int return type signifies the exit status of the program.
  - std::cout: Standard output stream, used for printing to the console.
  - std::endl: Ends the current line and flushes the output buffer

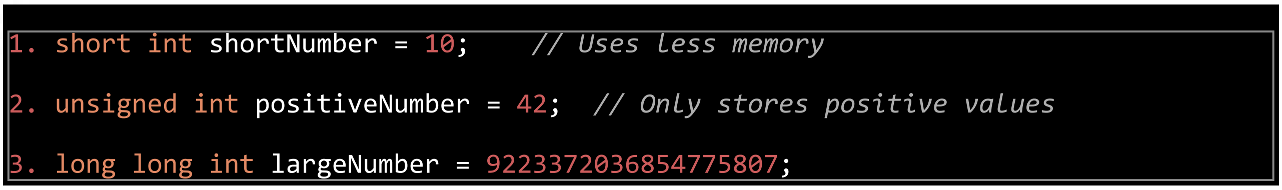
02. Data Types

C++ provides a wide variety of data types for different purposes, including primitive and user-defined data types.

- **Primitive Data Types**

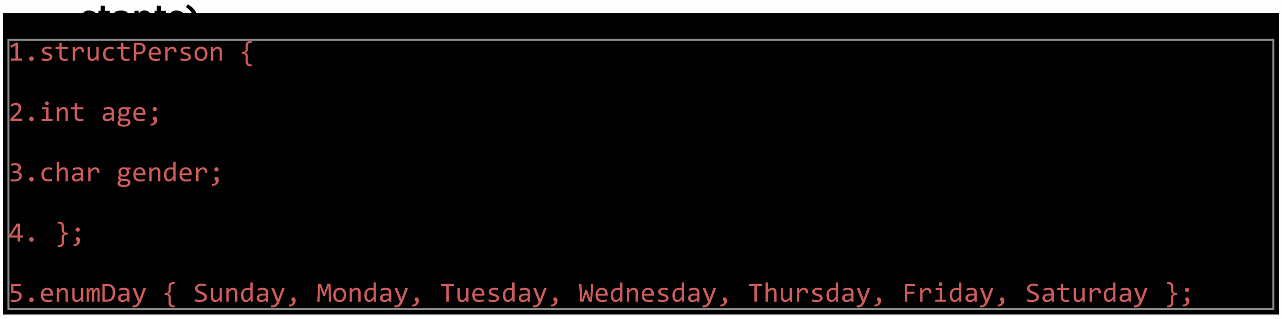
Type	Description	Example
int	Integer numbers	int age = 25;
float	Floating-point numbers	float weight = 60.5;
double	Double-precision float	double pi = 3.14159;
char	Single character	char grade = ‘A’;
bool	Boolean (True/ False)	bool isHappy = true;
void	Represents no type or void	Used in functions with no return value

- **Modifiers for Data Types:**
    - **Signed/Unsigned:** Modifies integer types to include negative numbers or only positive numbers.
    - **Short/Long:** Modifies integer types to reduce or extend their storage capacity.



```
short int shortNumber = 10;     // Uses less memory
unsigned int positiveNumber = 42;  // Only stores positive values
long long int largeNumber = 9223372036854775807;
```

- **User-Defined Data Types:**
    - **struct:** Used to group different data types together.
    - **enum:** Used to define an enumeration (a set of named integral constants)

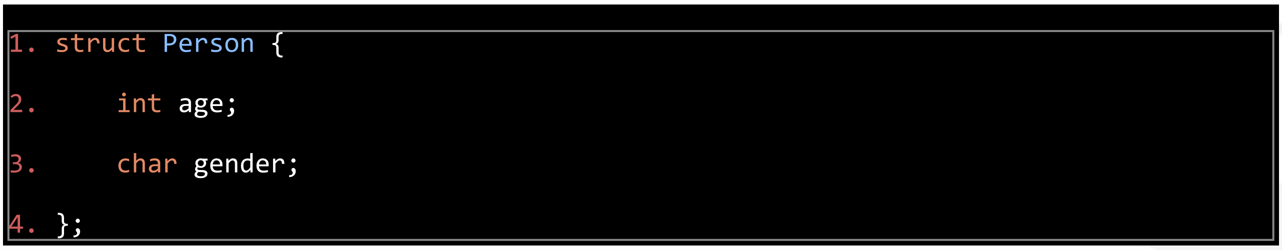


```
structPerson {
int age;
char gender;
 };
enumDay { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };
```

03. Variables and Constants

- **Variable Declaration:**

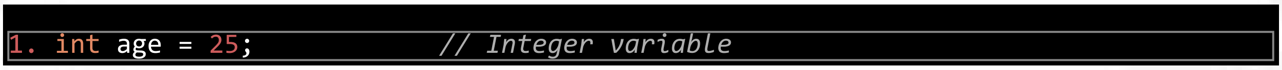
Variables in C++ must be declared before they are used, specifying their type.



```
struct Person {
    int age;
    char gender;
};
```

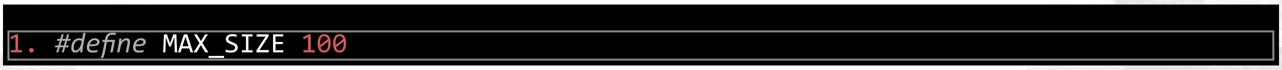
- **Constants:**

Constants in C++ are immutable and can be defined using the const keyword.



```
int age = 25;            // Integer variable
```

Alternatively, constants can also be defined using #define



```
#define MAX_SIZE 100
```

04. Operators in C++

C++ supports a wide variety of operators for performing operations on variables.

- **Arithmetic Operators:**

Operator	Description	Example
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	Division	a / b
%	Modulus (Remainder)	a % b

- **Relational Operators:**

Operator	Description	Example
==	Equal to	a == b
!=	Not equal to	a != b
>	Greater than	a > b
<	Less than	a < b
>=	Greater or equal	a >= b
<=	Less or equal	a <= b

- **Logical Operators:**

Operator	Description	Example
&&	Logical AND	a && b
`		`
!	Logical NOT	!a

- **Assignment Operators:**

Operator	Description	Example
=	Assign value	a = b
+=	Add and assign	a += b
-=	Subtract and assign	a -= b
*=	Multiply and assign	a *= b
/=	Divide and assign	a /= b

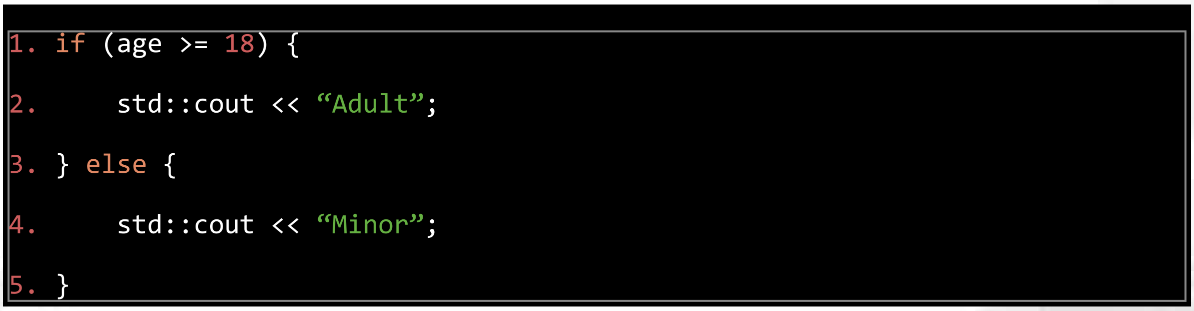
- **Increment/Decrement Operators:**

Operator	Description	Example
++a	Pre-increment (before use)	++a
a++	Post-increment (after use)	a++
--a	Pre-decrement (before use)	--a
a--	Post-decrement (after use)	a--

05. Control Structures

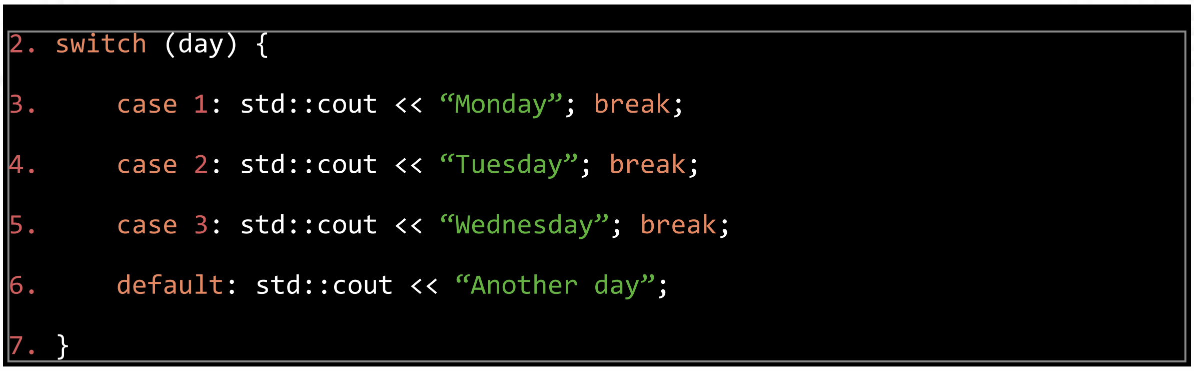
C++ provides control structures like conditional statements and loops to control the flow of execution in programs.

- **If-Else Statement:**



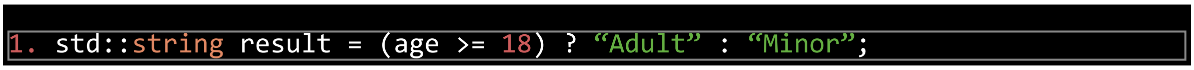
```
if (age >= 18) {
    std::cout << “Adult”;
} else {
    std::cout << “Minor”;
}
```

- **Switch Case:**



```cpp
switch (day) {
    case 1: std::cout << “Monday”; break;
    case 2: std::cout << “Tuesday”; break;
    case 3: std::cout << “Wednesday”; break;
    default: std::cout << “Another day”;
}
```

- **Ternary Operator:**

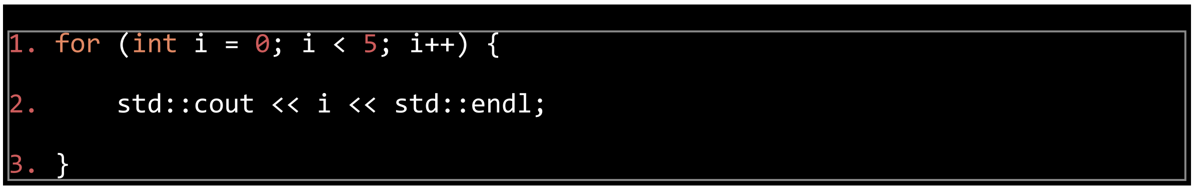


```cpp
std::string result = (age >= 18) ? “Adult” : “Minor”;
```

06. Loops

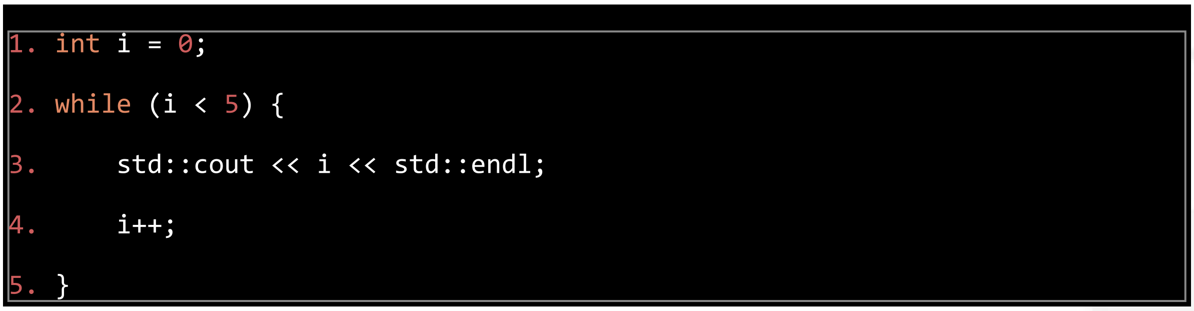
Loops allow repetitive execution of a block of code.

- **For Loop:**



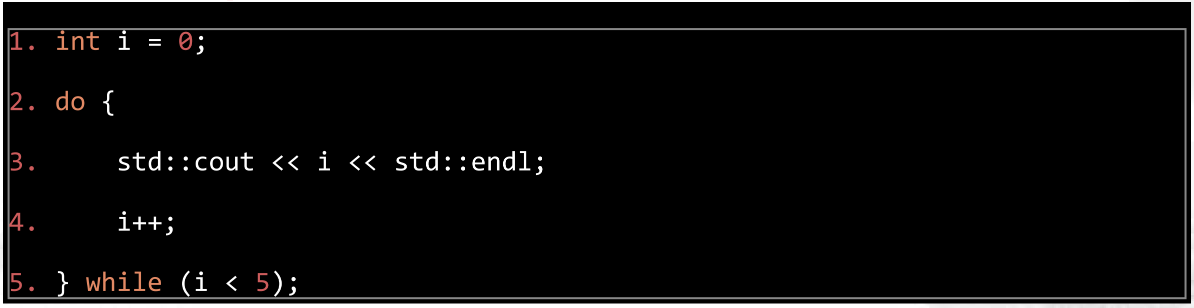
```cpp
for (int i = 0; i < 5; i++) {
    std::cout << i << std::endl;
}
```

- **While Loop:**



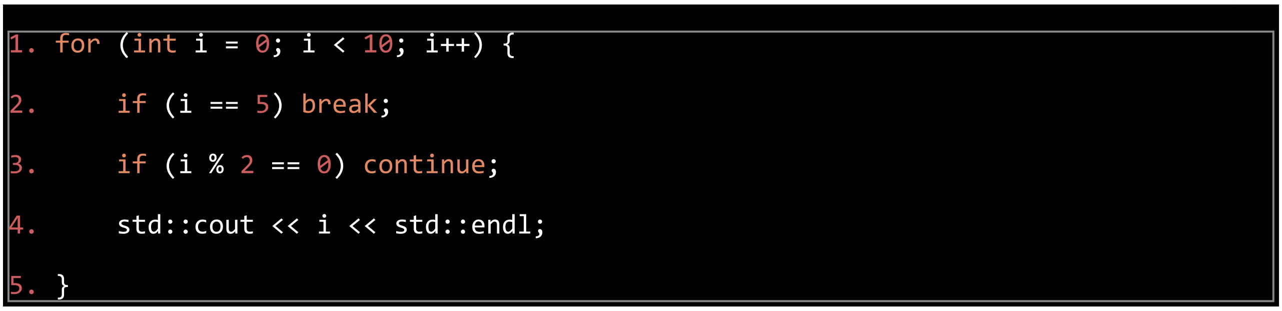
```cpp
int i = 0;
while (i < 5) {
    std::cout << i << std::endl;
    i++;
}
```

- **Do-While Loop:**



```cpp
int i = 0;
do {
    std::cout << i << std::endl;
    i++;
} while (i < 5);
```

- **Break and Continue:**
  - break: Exits a loop immediately.
  - continue: Skips the current iteration and moves to the next one.

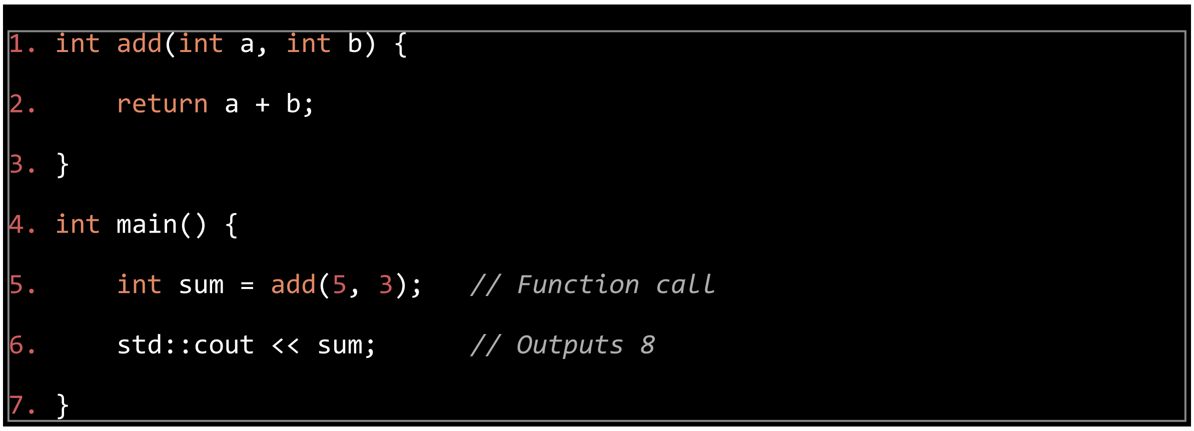


```cpp
for (int i = 0; i < 10; i++) {
    if (i == 5) break;
    if (i % 2 == 0) continue;
    std::cout << i << std::endl;
}
```

07. Functions in C++

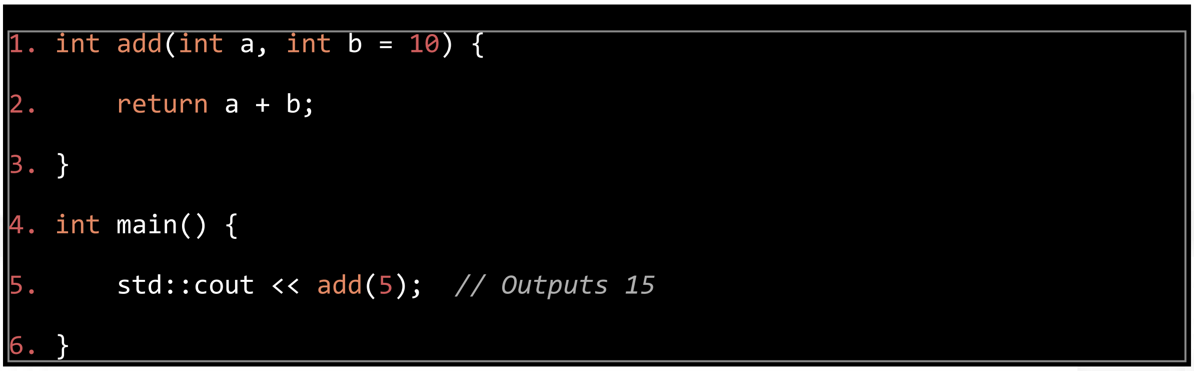
Functions allow code to be modular, reusable, and easier to understand.

- **Defining and Calling Functions:**



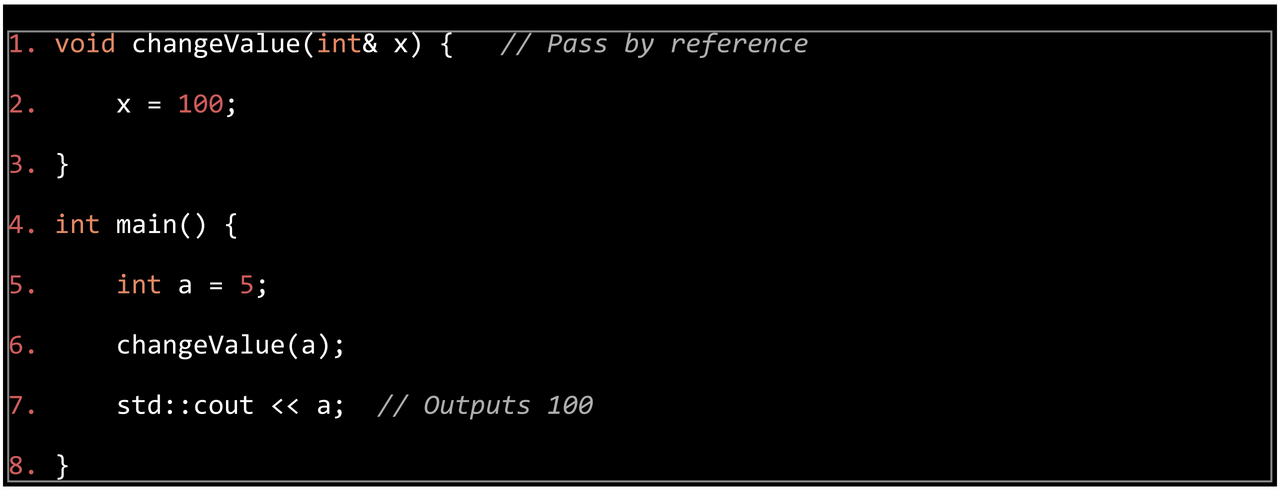
```cpp
int add(int a, int b) {
    return a + b;
}
int main() {
    int sum = add(5, 3);    // Function call
    std::cout << sum;       // Outputs 8
}
```

- **Default Parameters:**



```cpp
int add(int a, int b = 10) {
    return a + b;
}
int main() {
    std::cout << add(5);  // Outputs 15
}
```

- **Pass by Value and Reference:**
  - **Pass by Value:** The actual value is passed, changes to the parameter inside the function have no effect on the actual argument.
  - **Pass by Reference:** The reference of the variable is passed, changes to the parameter affect the actual argument

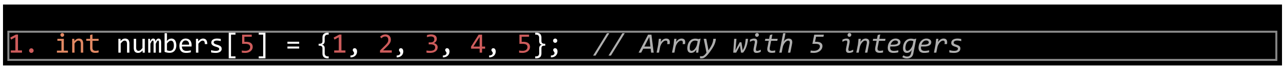


```cpp
void changeValue(int& x) {   // Pass by reference
    x = 100;
}
int main() {
    int a = 5;
    changeValue(a);
    std::cout << a;  // Outputs 100
}
```

08. Arrays

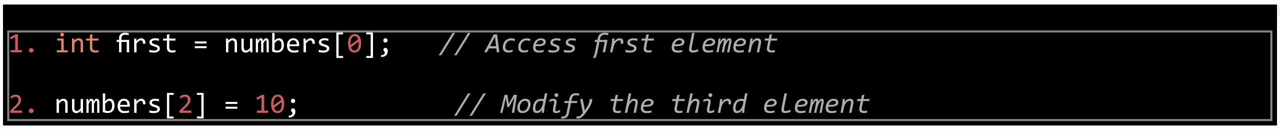
Arrays in C++ are used to store multiple values of the same data type.

- **Declaring Arrays:**



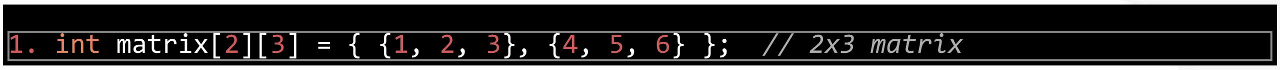
```cpp
int numbers[5] = {1, 2, 3, 4, 5};  // Array with 5 integers
```

- **Accessing Array Elements:**



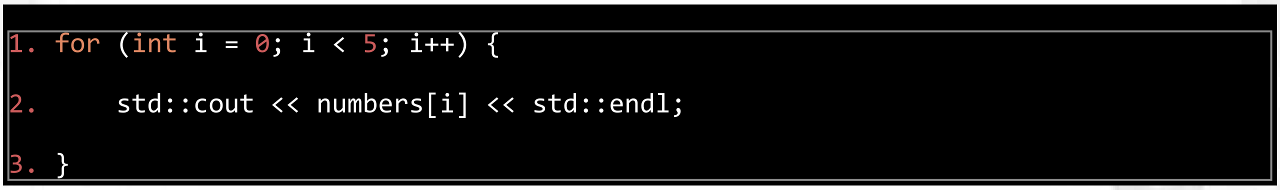
```cpp
int first = numbers[0];   // Access first element
numbers[2] = 10;          // Modify the third element
```

- **Multidimensional Arrays:**



```cpp
int matrix[2][3] = { {1, 2, 3}, {4, 5, 6} };  // 2x3 matrix
```

- **Iterating Arrays:**



```cpp
for (int i = 0; i < 5; i++) {
    std::cout << numbers[i] << std::endl;
}
```

09. Pointers and References

C++ provides powerful memory manipulation features through pointers and references.

- **Pointers:**
  - A pointer is a variable that stores the memory address of another variable

```
int a = 5;
int* p = &a;    // Pointer to integer ‘a’
 std::cout << *p;   // Dereferencing, outputs 5
```

  - & Operator: Gets the address of a variable.
  - * Operator: Dereferences the pointer, accessing the value at the

- **Pointer Arithmetic:**
  - You can perform arithmetic operations on pointers, such as incrementing them to access the next memory location.

```
int arr[3] = {10, 20, 30};
int* ptr = arr;
 std::cout << *(ptr + 1);   // Outputs 20
```

- **Null Pointers:**
  - A null pointer is a pointer that doesn’t point to any valid address.

```
int* ptr = nullptr;
```

- **References:**
  - References provide an alternative name for a variable. Unlike pointers, they cannot be null or reassigned after initialization.

```
int a = 10;
int&ref = a;    // Reference to variable ‘a’
ref = 20;         // Changes the value of ‘a’
```

10. Strings

C++ supports strings through both the C-style character array (char[]) and the std::string class.

- **C-Style Strings:**

```
char greeting[] = “Hello, World!”;
 std::cout << greeting;
```

- std::string Class:

```
#include<string>
 std::string name = “John”;
 std::cout <<”Hello, “<< name <<”!”;
```

- **Common String Operations:**

```
 std::string firstName = “John”;
 std::string lastName = “Doe”;
 std::string fullName = firstName + “ “ + lastName;
```

- Length:

```
 std::string str = “Hello”;
 std::cout << str.length();  // Outputs 5
```

- Substring:

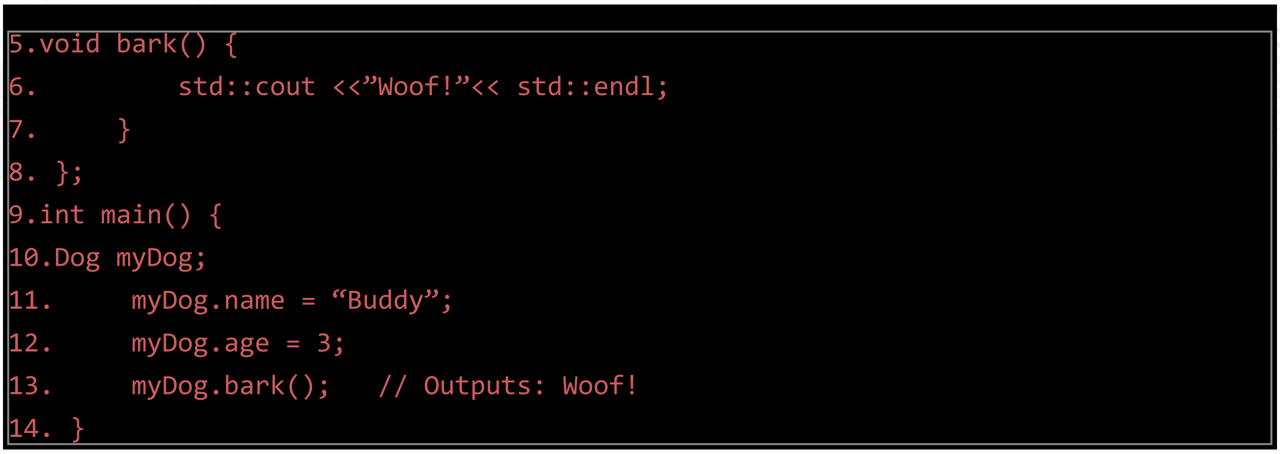
```
 std::string text = “Hello, World!”;
 std::stringsub = text.substr(0, 5);  // Outputs “Hello”
```

11. Object-Oriented Programming (OOP)

C++ supports the principles of OOP, such as encapsulation, inheritance, and

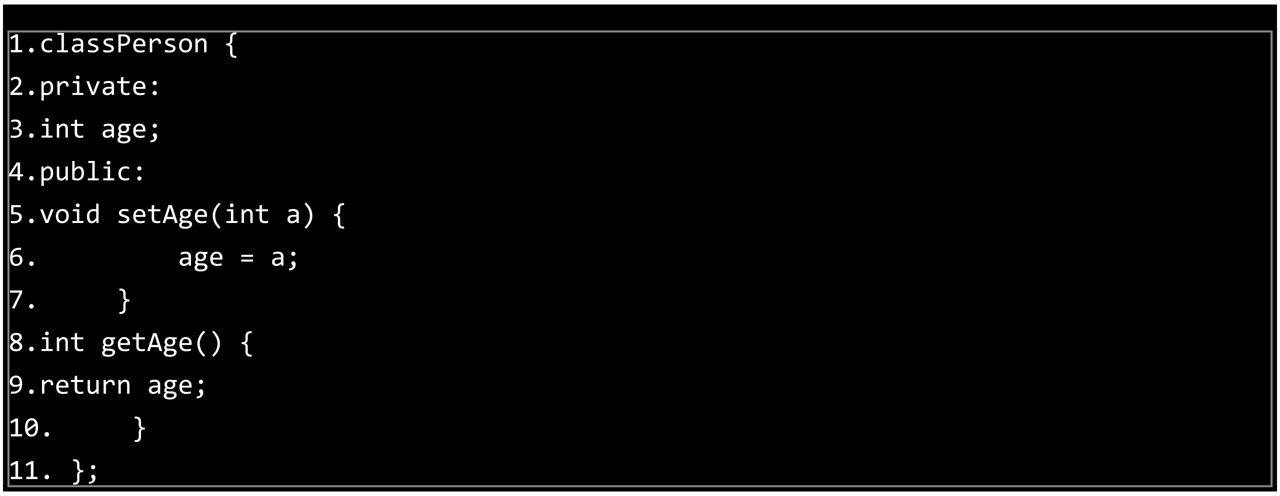
- **Classes and Objects:**
  - A class is a blueprint for creating objects.

```
classDog {
public:
     std::string name;
int age;
```



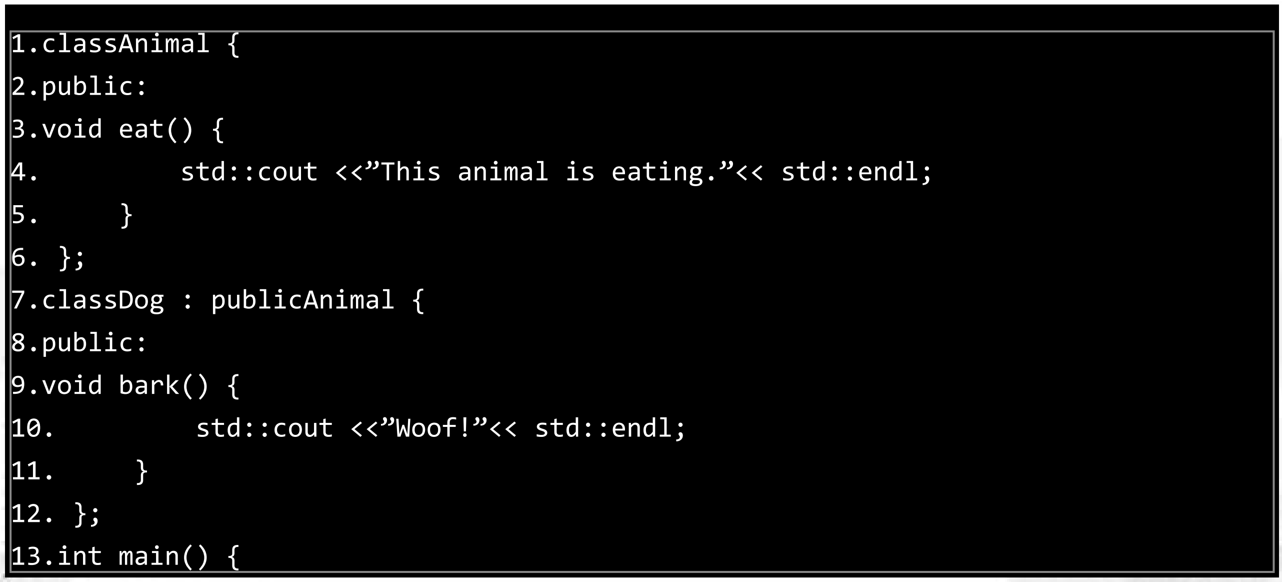
```
void bark() {
         std::cout <<”Woof!”<< std::endl;
     }
 };
int main() {
Dog myDog;
        myDog.name = “Buddy”;
        myDog.age = 3;
        myDog.bark();   // Outputs: Woof!
 }
```

- **Encapsulation:**
    - **Data and functions that manipulate that data are encapsulated within a class.**



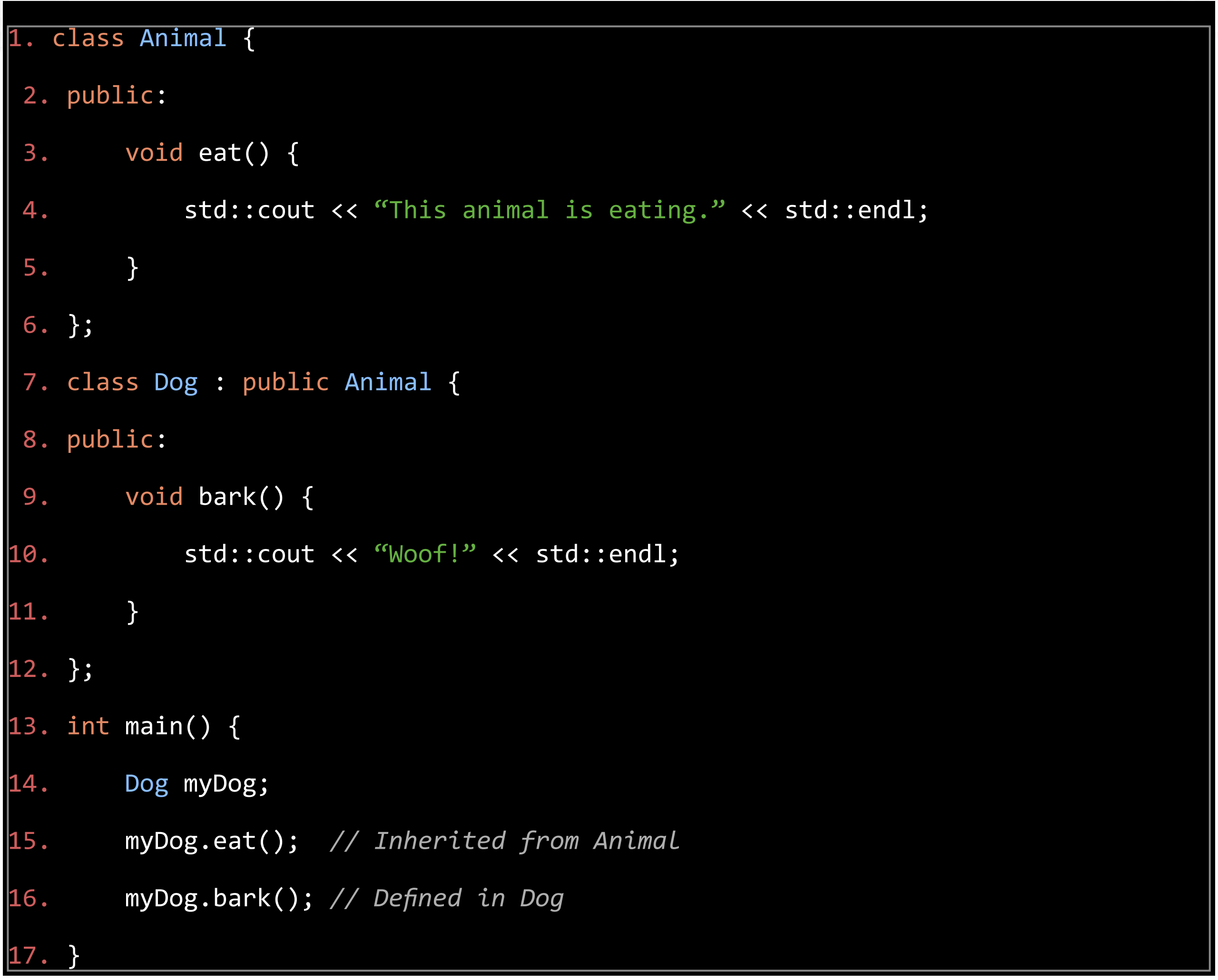
```
classPerson {
private:
int age;
public:
void setAge(int a) {
         age = a;
     }
int getAge() {
return age;
        }
 };
```

- **Inheritance:**
    - **Inheritance allows one class to inherit properties and methods from another class.**



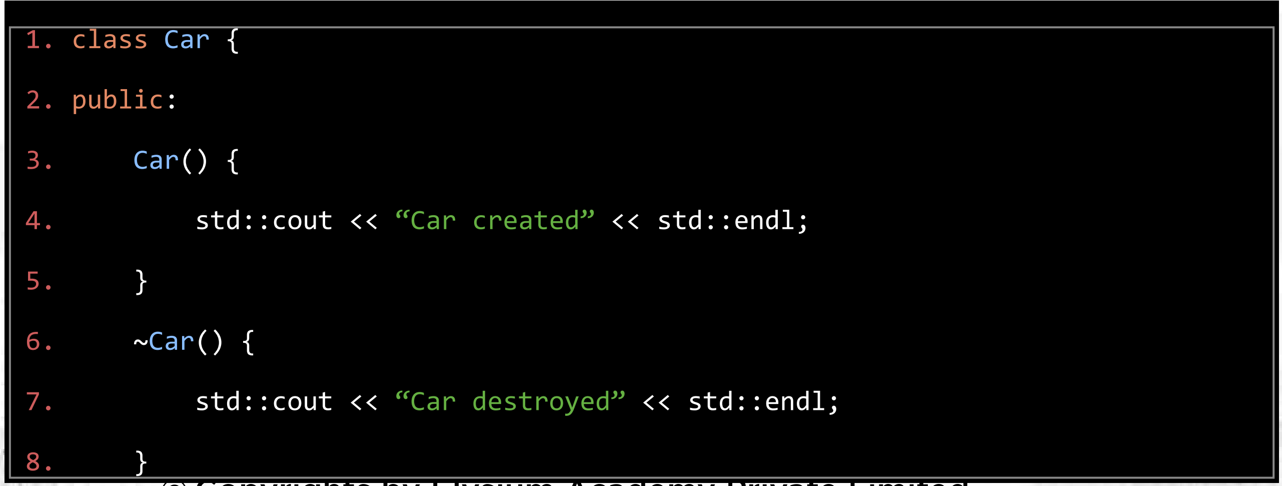
```
classAnimal {
public:
void eat() {
         std::cout <<”This animal is eating.”<< std::endl;
     }
 };
classDog : publicAnimal {
public:
void bark() {
         std::cout <<”Woof!”<< std::endl;
        }
 };
int main() {
```

- **Polymorphism:**
  - Polymorphism allows one interface to be used for a general class of actions, typically achieved through function overloading and over-



```cpp
class Animal {
public:
    void eat() {
        std::cout << “This animal is eating.” << std::endl;
    }
};
class Dog : public Animal {
public:
    void bark() {
        std::cout << “Woof!” << std::endl;
    }
};
int main() {
    Dog myDog;
    myDog.eat();  // Inherited from Animal
    myDog.bark(); // Defined in Dog
}
```

- **Constructors and Destructors:**
  - **Constructor:** A special function that initializes an object when it’s created.
  - **Destructor:** A special function that cleans up an object when it’s destroyed



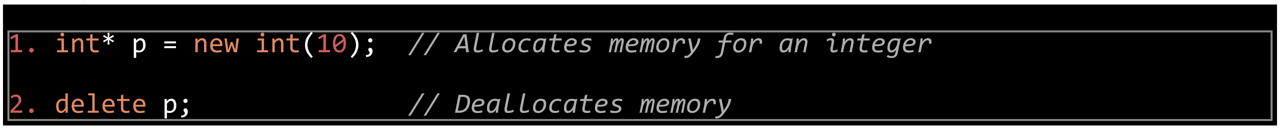
```cpp
class Car {
public:
    Car() {
        std::cout << “Car created” << std::endl;
    }
    ~Car() {
        std::cout << “Car destroyed” << std::endl;
    }
```

```cpp
};
```

12. Dynamic Memory Management

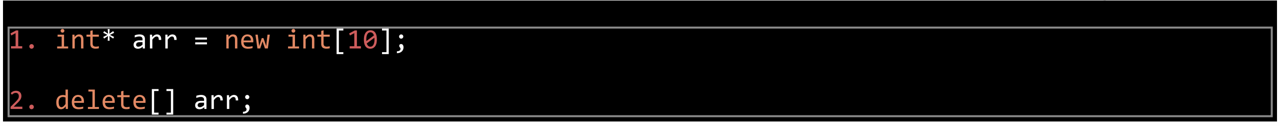
C++ allows you to manually manage memory allocation and deallocation using new and delete.

- **Dynamic Allocation:**



```cpp
int* p = new int(10);   // Allocates memory for an integer
delete p;               // Deallocates memory
```

- new[] and delete[]: Used to allocate and deallocate arrays.

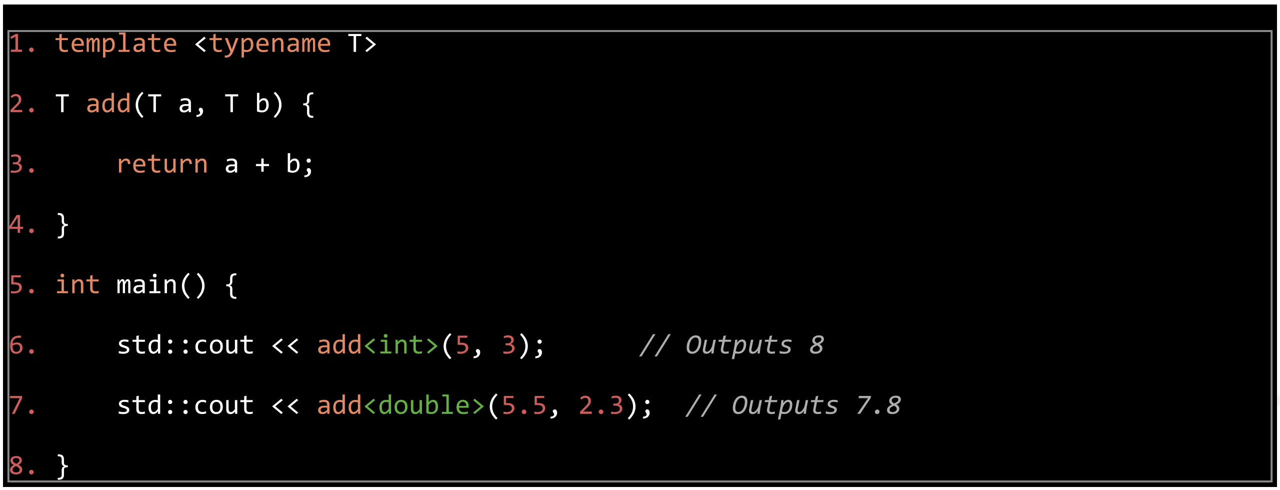


```cpp
int* arr = new int[10];
delete[] arr;
```

13. Templates

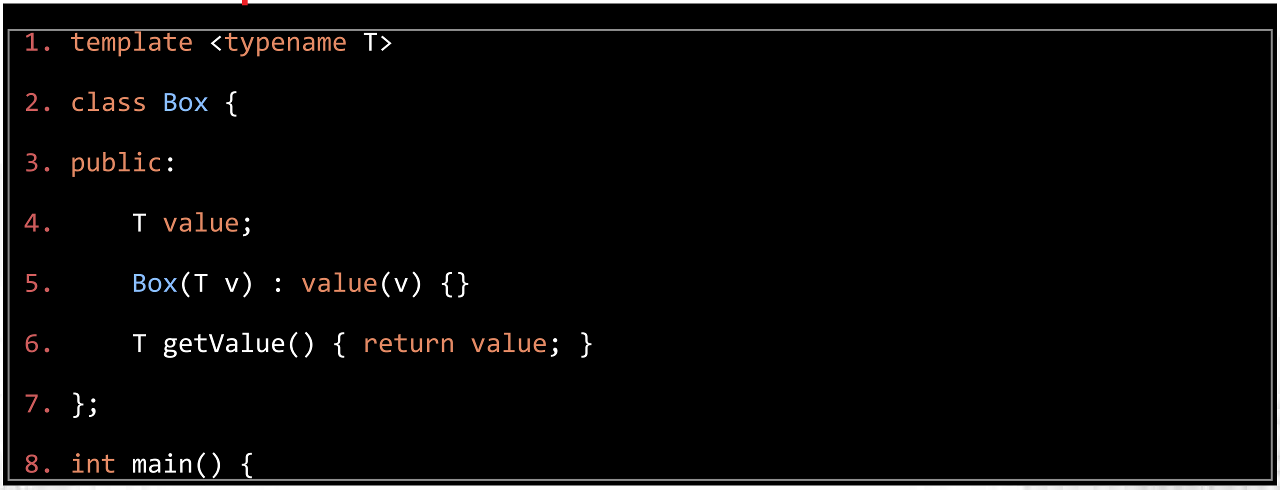
Templates enable generic programming by allowing you to write functions and classes that work with any data type.

- **Function Template:**

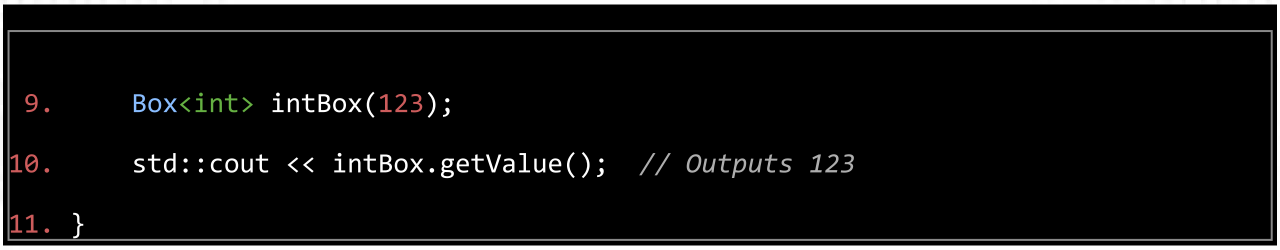


```cpp
template <typename T>
T add(T a, T b) {
    return a + b;
}
int main() {
    std::cout << add<int>(5, 3);      // Outputs 8
    std::cout << add<double>(5.5, 2.3);  // Outputs 7.8
}
```

- **Class Template:**



```cpp
template <typename T>
class Box {
public:
    T value;
    Box(T v) : value(v) {}
    T getValue() { return value; }
};
int main() {
```

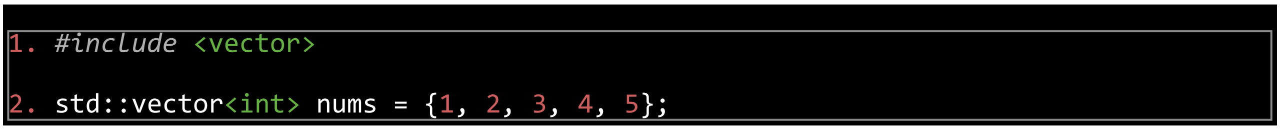


```
    Box<int> intBox(123);
    std::cout << intBox.getValue();  // Outputs 123
}
```

14. Standard Template Library (STL)

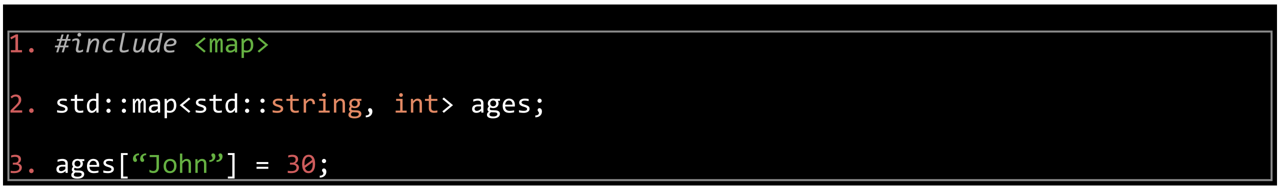
The STL provides a collection of useful data structures and algorithms.

- **Common Containers:**
  - **Vector:** Dynamic array.



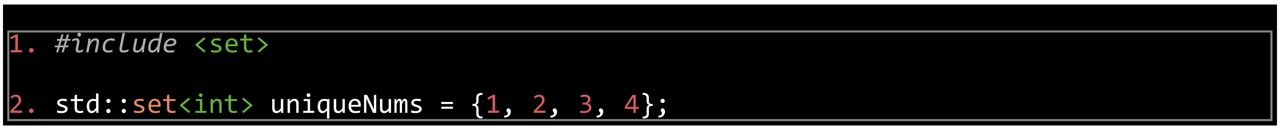
```
#include <vector>
std::vector<int> nums = {1, 2, 3, 4, 5};
```

  - **Map:** Key-value pairs.



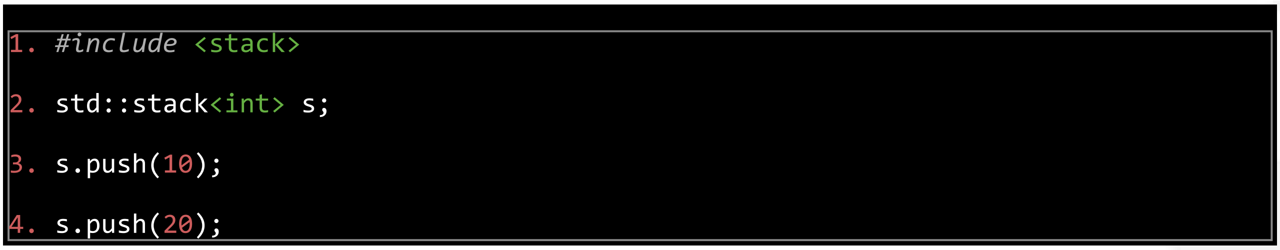
```
#include <map>
std::map<std::string, int> ages;
ages[“John”] = 30;
```

  - **Set:** Unique collection of elements



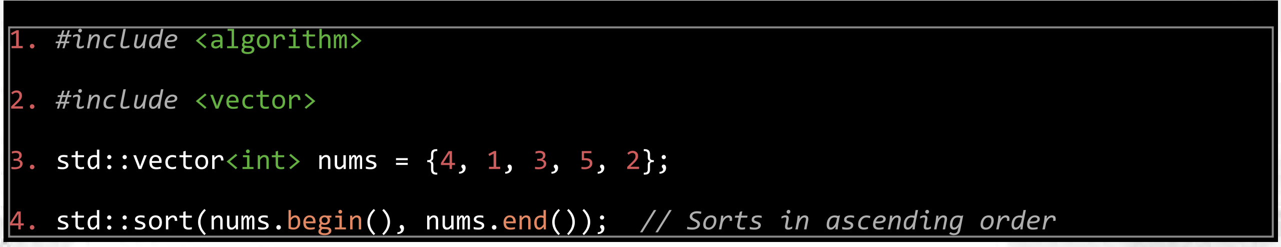
```
#include <set>
std::set<int> uniqueNums = {1, 2, 3, 4};
```

  - **Stack:** LIFO (Last In, First Out).



```
#include <stack>
std::stack<int> s;
s.push(10);
s.push(20);
```

- **Common Algorithms:**
  - **STL provides algorithms like sort(), find(), reverse(), etc.**



```
#include <algorithm>
#include <vector>
std::vector<int> nums = {4, 1, 3, 5, 2};
std::sort(nums.begin(), nums.end());  // Sorts in ascending order
```

15. Exception Handling

C++ provides support for exception handling using try, catch, and throw blocks.

- **Try-Catch Block:**

```
try {
    int result = 10 / 0;  // Division by zero
} catch (const std::exception& e) {
    std::cout << “Error: “ << e.what() << std::endl;
}
```

- **Throwing Exceptions:**

```
throw std::invalid_argument(“Invalid argument passed!”);
```

16. Namespaces

Namespaces are used to organize code and avoid name collisions.

```
namespace Math {
    int add(int a, int b) {
        return a + b;
    }
}
int main() {
    std::cout << Math::add(5, 3);  // Outputs 8
}
```

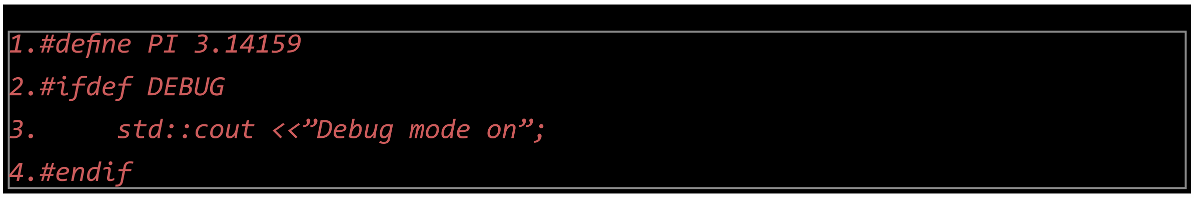
- **Using using Keyword:**

```
using namespace Math;
std::cout << add(5, 3);  // No need for Math:: prefix
```

17. Preprocessor Directives

C++ provides several preprocessor directives for code management.

- **#include:** Includes the contents of a file.
- **#define:** Defines constants or macros.
- **#ifdef / #ifndef:** Conditional compilation



```
#define PI 3.14159
#ifdef DEBUG
     std::cout <<"Debug mode on";
#endif
```

18. C++ Best Practices

- **Follow Naming Conventions:**
  - **Use camelCase for variables and functions.**
  - **Use PascalCase for class names.**
- **Use RAII for Memory Management:**
  - Resource Acquisition Is Initialization (RAII) ensures that resources are properly released.
- **Prefer std::string Over C-Style Strings:**
  - std::string handles memory automatically and is easier to use.
- **Use Smart Pointers:**
  - Use std::shared_ptr and std::unique_ptr for automatic memory management and to avoid memory leaks.
- **Avoid Magic Numbers:**
  - Define meaningful constants instead of using raw numbers in your code.
- **Always Check for Pointer Validity:**
  - Ensure pointers are valid before dereferencing them.
- **Write Modular and Reusable Code:**
  - Break large functions into smaller, reusable ones for readability and maintainability.

19. Conclusion

C++ is a robust and versatile language that provides excellent control over system resources, making it ideal for developing high-performance applications. This comprehensive covered fundamental and advanced concepts, from basic syntax and data types to object-oriented programming, memory management, templates, and the Standard Template Library (STL).
As you continue developing in C++, adhering to best practices such as effective memory management, code modularity, and leveraging STL and RAII principles will ensure your C++ programs are efficient, maintainable, and scalable. Happy coding!

Click Here To Find Out More

ELYSIUM SPARK NOTES

ELYSIUM ACADEMY®
Milestone of Cognizance

Thank you

For Your Learning Today

elysiumacademy.org | info@elysiumacademy.org

Scan Here for More
Spark Notes