## MQF 633 C++ FOR FINANCIAL ENGINEERING

# Lecture 2: C++ Data Types, Variable and Bacis Syntax

## Part I: Understanding C++ Data Types & Variable

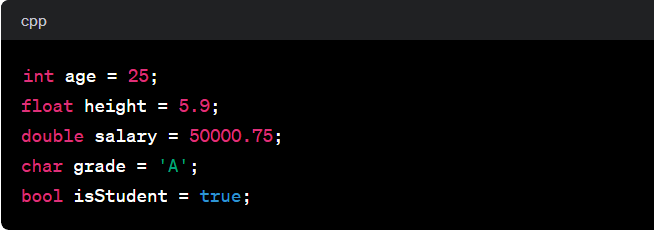
### Overview of Data Types

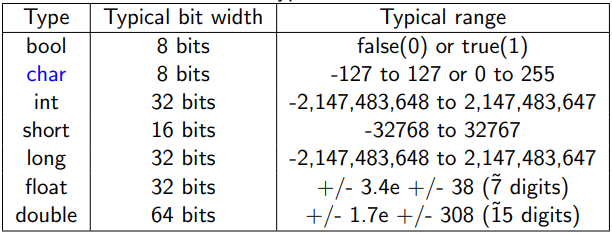
In C++, data types are used to define the type of data that a variable can hold. It helps in allocating memory and performing operations on variables. C++ supports a variety of data types.

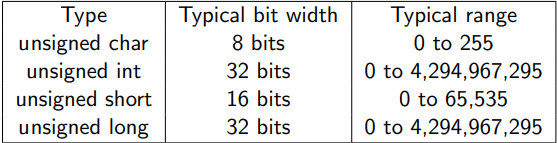
### Fundamental Data Types

* int: Integer type
* float: Floating-point type
* double: Double-precision floating-point type
* char: Character type
* bool: Boolean type
* long: 32bit integer type
* short: 16bit integer type
* unsigned int:

…

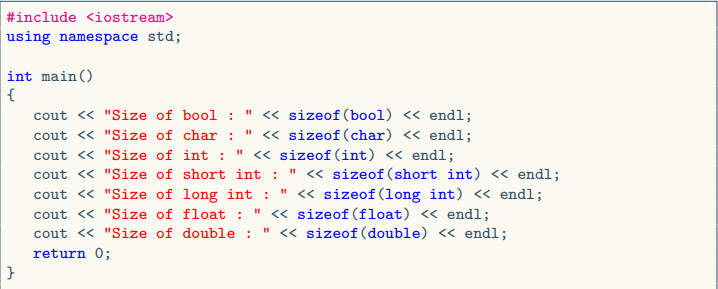






Note: 1 byte = 8 bits

Example: Actual sizeof() Data Types sizeof.cpp, refering to data\_type.cpp.



Some additional notes:

In C++, long and int are different data types, and they have different sizes. Both are used to represent integer values, but their sizes can vary depending on the compiler and the platform. Here are the typical sizes for these data types on a 32-bit and 64-bit system:

On a 32-bit system:

* int: 4 bytes
* long: 4 bytes

On a 64-bit system:

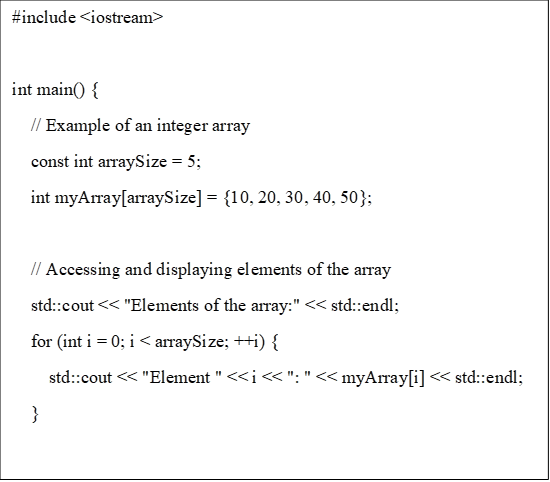
* int: 4 bytes
* long: 8 bytes

In C++, the sizes of these data types are implementation-dependent, meaning they can vary from one compiler to another and from one system to another. The C++ standard only specifies the minimum size of each data type.

If you need a guaranteed size regardless of the platform or compiler, you can use fixed-size integer types provided by the <cstdint> header, such as int32\_t and int64\_t for 32-bit and 64-bit signed integers, respectively.

### Derived Data Types

Array example: Collection of similar data types, int, double, char ... etc, referr to array.cpp



Explanation

We declare an integer array named myArray with a size of 5. The elements of the array are initialized using the curly braces {}.

We use a for loop to iterate through the elements of the array and display them.

We modify the value of the element at index 2 (third element) to 35. We display the modified array.

This is a basic example, and arrays in C++ can be used for various purposes, such as storing collections of data, implementing algorithms, and more. Keep in mind that C++ arrays have a fixed size, and their size is specified at the time of declaration. If you need a dynamic-sized array, you might want to consider using a container like std::vector from the C++ Standard Library.

### Pointer: Stores memory address of another variable

In C++, a pointer is a variable that holds the memory address of another variable. It allows indirect access to the value stored in a variable by referencing its memory address. Here's a brief definition and an example of using pointers in C++: A pointer is a variable that stores the memory address of another variable. It is declared using the data type of the variable it points to, followed by an asterisk (\*).

Example of pointer type: referring to pointer.cpp

#include <iostream>

int main() {

// Declare a variable

int number = 42;

// Declare a pointer to an integer and initialize it with the address of 'number'

int \*pointerToNumber = &number;

// Display the value of 'number' and the memory address it occupies

std::cout << "Value of 'number': " << number << std::endl;

std::cout << "Memory address of 'number': " << &number << std::endl;

// Display the value of 'pointerToNumber' (memory address)

std::cout << "Value of 'pointerToNumber' (memory address): " << pointerToNumber << std::endl;

// Access the value of 'number' through the pointer and display it

std::cout << "Value of 'number' accessed through the pointer: " << \*pointerToNumber << std::endl;

return 0;

}

Explanation:

We declare an integer variable named number and initialize it with the value 42.

We declare a pointer to an integer (int \*pointerToNumber) and initialize it with the address of the number variable using the address-of operator &.

We display the value of number and its memory address using std::cout.

We display the memory address stored in the pointer pointerToNumber.

We access the value of number through the pointer using the dereference operator \* and display it.

This example demonstrates the basic usage of pointers, including declaration, initialization, accessing memory addresses, and dereferencing to access the value stored at a memory location.

## **Reference: Another name for an existing variable**

In C++, a reference is an alias or an alternative name for an existing variable. It provides a way to access the value of a variable indirectly through its reference. Here's a definition and an example of using references in C++:

Reference Definition: A reference is an alias or an alternative name for an existing variable. It is declared using the & symbol after the data type.

Example: refer to reference.cpp

#include <iostream>

int main() {

// Declare a variable

int number = 42;

// Declare a reference to an integer and initialize it with the variable 'number'

int& referenceToNumber = number;

// Display the value of 'number' and the value through the reference

std::cout << "Value of 'number': " << number << std::endl;

std::cout << "Value through the reference: " << referenceToNumber << std::endl;

// Modify the value through the reference

referenceToNumber = 99;

// Display the modified value of 'number'

std::cout << "Modified value of 'number': " << number << std::endl;

return 0;

}

### Pointer VS Reference

Pointer:

* Useful for dynamic memory allocation (e.g., with new and delete operators). It can be used to implement data structures and algorithms.

Reference:

* Convenient for passing variables by reference to functions.
* Often used to create aliases for existing variables in a clear and concise manner.

In summary, both pointers and references offer indirect access to variables, but pointers provide more flexibility in terms of reassignment and nullability, while references offer simplicity and are commonly used in function parameter passing. The choice between pointers and references depends on the specific requirements of the task at hand.

Difference:

1. Pointers need to be explicitly initialized with the memory address of a variable, References must be initialized when declared and cannot be changed to refer to a different variable after initialization.

int \*ptr = &variable;

int \*ptr;

ptr = &variable;

int &ref = variable;

1. Pointers can be reassigned to point to different memory locations. References, once initialized, cannot be reassigned to refer to another variable.

int \*ptr = &variable;

ptr = &variable2;

int &ref = variable;

1. Pointers can be set to nullptr to represent a null or undefined state. References must always be initialized and cannot be null.

int \*ptr = nullptr;

// Error: references must be initialized

int &ref;

1. Pointer arithmetic operation.

int \*ptr = &variable;

ptr ++;

#include <iostream>

int main() {

int arr[] = {10, 20, 30, 40, 50};

int \*ptr = arr;

// Pointer arithmetic

ptr = ptr + 2; // Increment by 2 elements

std::cout << "Value at the incremented position: " << \*ptr << std::endl;

// Array indexing using pointer arithmetic

int thirdElement = \*(ptr + 1); // Access the third element

std::cout << "Value of the third element: " << thirdElement << std::endl;

// Comparison of pointers

int \*ptr2 = arr + 2;

if (ptr == ptr2) {

std::cout << "Pointers are equal." << std::endl;

} else {

std::cout << "Pointers are not equal." << std::endl;

}

return 0;

}

## User-Defined Data Types

In C++, user-defined types allow programmers to create their own data types, providing a way to organize and structure data in a more meaningful and modular way. The primary user-defined types in C++ include:

* Struct: Groups variables of different types under a single name
* Class: Similar to struct but allows encapsulation and inheritance
* Enumeration: Defines a set of named integer constants

Strut example

#include <iostream>

#include <string>

// Define a struct named Person

struct Person {

// Data members

std::string name;

int age;

double height;

};

int main() {

// Create an instance of the Person struct

Person person1;

// Assign values to the struct members

person1.name = "John Doe";

person1.age = 25;

person1.height = 5.9;

// Display the information using the struct members

std::cout << "Person Information:" << std::endl;

std::cout << "Name: " << person1.name << std::endl;

std::cout << "Age: " << person1.age << " years" << std::endl;

std::cout << "Height: " << person1.height << " feet" << std::endl;

return 0;

}

Explanation:

1. We define a struct named Person with three data members: name (string), age (integer), and height (double).
2. In the main function, we create an instance of the Person struct named person1.
3. We assign values to the data members of person1.
4. We use the struct members to display the information about the person.
5. Structs are useful for grouping related data members together, creating user-defined data types, and organizing data in a more meaningful way. They provide a convenient way to represent and work with complex data structures.

Class example:

#include <iostream>

#include <string>

// Define a class named Student

class Student {

public:

// Public data members

std::string name;

int age;

double GPA;

// Public member function to display student information

void displayInfo() {

std::cout << "Student Information:" << std::endl;

std::cout << "Name: " << name << std::endl;

std::cout << "Age: " << age << " years" << std::endl;

std::cout << "GPA: " << GPA << std::endl;

}

};

int main() {

// Create an instance of the Student class

Student student1;

// Assign values to the class members

student1.name = "Alice Smith";

student1.age = 20;

student1.GPA = 3.75;

// Call the member function to display student information

student1.displayInfo();

return 0;

}

Explanation:

1. We define a class named Student with three public data members: name (string), age (integer), and GPA (double).
2. Inside the class, we declare a public member function displayInfo() to display the information about the student using the class members.
3. In the main function, we create an instance of the Student class named student1.
4. We assign values to the data members of student1.
5. We call the displayInfo() member function to display the information about the student.
6. Classes are a fundamental concept in object-oriented programming (OOP) and provide a way to model real-world entities with attributes (data members) and behaviors (member functions).

Enum example

#include <iostream>

// Define an enumeration named Days

enum Days {

Sunday,

Monday,

Tuesday,

Wednesday,

Thursday,

Friday,

Saturday

};

int main() {

// Declare a variable of type Days

Days today = Wednesday;

// Switch statement using the enum values

switch (today) {

case Sunday:

std::cout << "It's Sunday!" << std::endl;

break;

case Monday:

std::cout << "It's Monday!" << std::endl;

break;

case Tuesday:

std::cout << "It's Tuesday!" << std::endl;

break;

case Wednesday:

std::cout << "It's Wednesday!" << std::endl;

break;

case Thursday:

std::cout << "It's Thursday!" << std::endl;

break;

case Friday:

std::cout << "It's Friday!" << std::endl;

break;

case Saturday:

std::cout << "It's Saturday!" << std::endl;

break;

}

return 0;

}

### Typedef and Using

Allows creating an alias for an existing type aliases. refer to typedef\_using.cpp

#include <iostream>

// Original data type

typedef double Distance;

int main() {

// Using the typedef alias

Distance length = 10.5;

Distance width = 5.2;

// Calculating the area

Distance area = length \* width;

// Displaying the result

std::cout << "Length: " << length << std::endl;

std::cout << "Width: " << width << std::endl;

std::cout << "Area: " << area << std::endl;

return 0;

}

In this example, typedef double Distance; creates an alias Distance for the double data type. The rest of the program uses Distance to declare variables, making the code more readable and abstracting away the specific data type. Note that starting from C++11, the using keyword is often preferred over typedef for creating type

#include <iostream>

// Equivalent to typedef double Distance;

using Distance = double;

int main() {

// Using the alias

Distance length = 10.5;

Distance width = 5.2;

// Calculating the area

Distance area = length \* width;

// Displaying the result

std::cout << "Length: " << length << std::endl;

std::cout << "Width: " << width << std::endl;

std::cout << "Area: " << area << std::endl;

return 0;

}

Const modifier

* Use const for variables that should not be modified after initialization.
* Use const with pointers and references to enforce read-only access to data.

By using const appropriately, you make your code more robust and expressive, helping catch potential errors at compile-time. Refer to Const\_example.cpp

#include <iostream>

int main() {

// Constant variable

const int maxAttempts = 3;

// Normal variable

int score = 100;

// Constant pointer to an integer

const int\* constPointer = &score;

// Constant reference to an integer

const int& constReference = score;

// Attempting to modify the constant variable will result in a compilation error

// maxAttempts = 4; // Error: assignment of read-only variable 'maxAttempts'

// Attempting to modify the value through the constant pointer or reference is not allowed

// (\*constPointer)++; // Error: increment of read-only location

// constReference++; // Error: increment of read-only reference

// Displaying values

std::cout << "Max Attempts: " << maxAttempts << std::endl;

std::cout << "Score: " << score << std::endl;

return 0;

}

## Variables

A variable is a named storage (in memory) that our programs can manipulate, In C++, each variable has a type, the type of the variable determines:

* The size and layout of variable’s memory,
* How this information is interpreted (integer, floating point, etc)
* The range of values that can be stored, and
* The set of operations that can be applied
* The value of the variable is the information stored in this memory location
* Data type of the variable tell the amount of memory space the variable needs and how to interpret the memory

### Variable Name

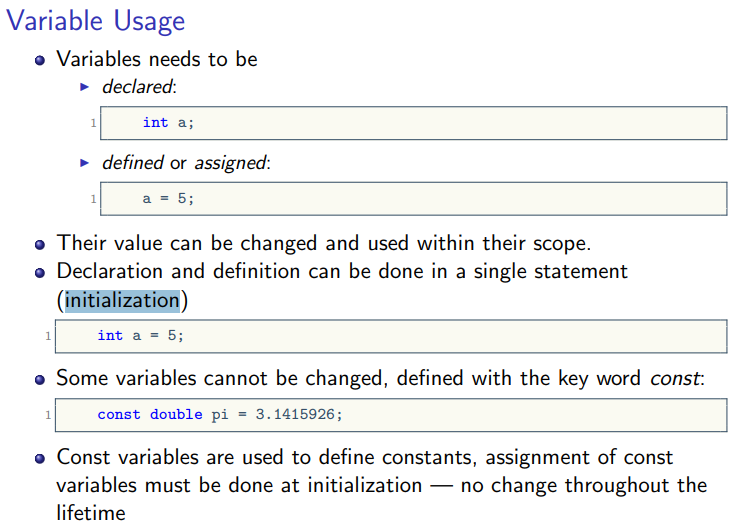
Variable name cannot start with a digit, e.g., 1a is not a valid variable name — compilation error

Variable names cannot contain whitespaces or special characters like: !, #, %, etc.

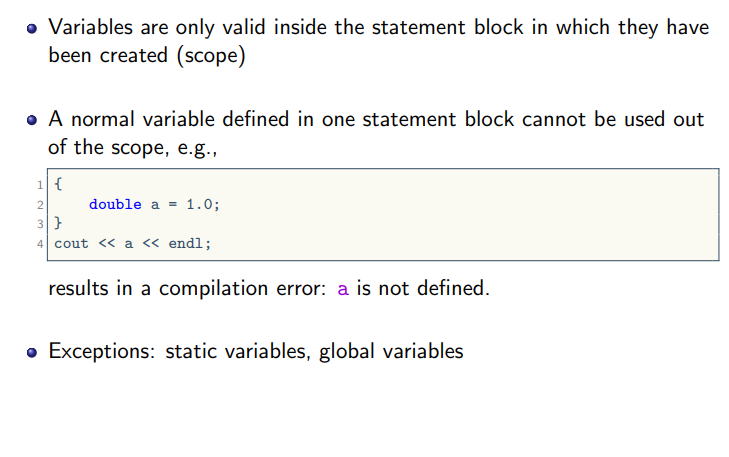
Naming conventions: rules for choosing the character sequence to be used for identifiers which denote variables, types, functions, etc.

* Multiple-word identifier format: two\_words, TWO\_WORDS, TwoWords,
* twoWords, etc.
* Good to be consistent for readability and code analysis tools
* Not easy to enforce consistency throughout the whole project
* See <https://en.wikipedia.org/wiki/Naming_convention> (programming)

### Define Variable

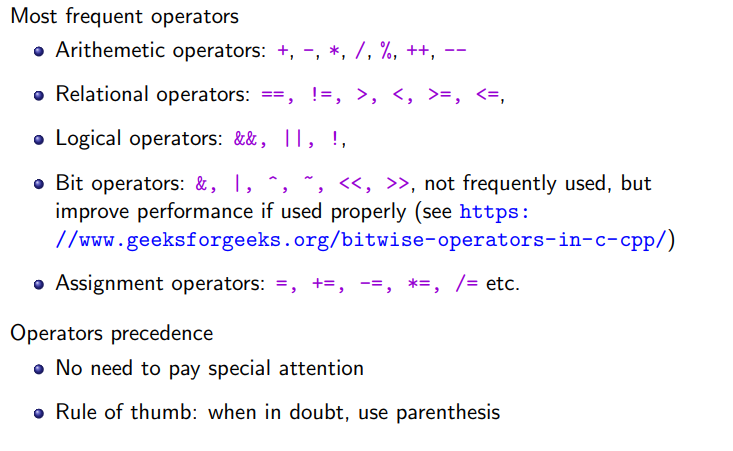


### Variable Scope



### Operator

refer to operator.cpp



Example Arithmetic operator

#include <iostream>

int main() {

int a = 10, b = 5;

// Addition

int sum = a + b;

// Subtraction

int difference = a - b;

// Multiplication

int product = a \* b;

// Division

int quotient = a / b;

// Modulus (remainder)

int remainder = a % b;

std::cout << "Sum: " << sum << std::endl;

std::cout << "Difference: " << difference << std::endl;

std::cout << "Product: " << product << std::endl;

std::cout << "Quotient: " << quotient << std::endl;

std::cout << "Remainder: " << remainder << std::endl;

return 0;

}

Example Relational Operators

#include <iostream>

int main() {

int x = 5, y = 10;

// Equal to

bool isEqual = (x == y);

// Not equal to

bool notEqual = (x != y);

// Greater than

bool greaterThan = (x > y);

// Less than

bool lessThan = (x < y);

// Greater than or equal to

bool greaterOrEqual = (x >= y);

// Less than or equal to

bool lessOrEqual = (x <= y);

std::cout << "Equal: " << isEqual << std::endl;

std::cout << "Not Equal: " << notEqual << std::endl;

std::cout << "Greater Than: " << greaterThan << std::endl;

std::cout << "Less Than: " << lessThan << std::endl;

std::cout << "Greater or Equal: " << greaterOrEqual << std::endl;

std::cout << "Less or Equal: " << lessOrEqual << std::endl;

return 0;

}

Logical Operators

#include <iostream>

int main() {

bool a = true, b = false;

// Logical AND

bool logicalAnd = (a && b);

// Logical OR

bool logicalOr = (a || b);

// Logical NOT

bool logicalNot = !a;

std::cout << "Logical AND: " << logicalAnd << std::endl;

std::cout << "Logical OR: " << logicalOr << std::endl;

std::cout << "Logical NOT: " << logicalNot << std::endl;

return 0;

}

## Part II: C++ Basic Syntax

### Flow control using if-else statement

#include <iostream>

int main() {

// Declare a variable to store the user input

int number;

// Prompt the user to enter a number

std::cout << "Enter an integer: ";

std::cin >> number;

// Check if the number is even or odd

if (number % 2 == 0) {

std::cout << number << " is an even number." << std::endl;

} else {

std::cout << number << " is an odd number." << std::endl;

}

return 0;

}

### Using a switch statement

#include <iostream>

int main() {

// Declare a variable to store the user input

int dayNumber;

std::cout << "Enter a number (1-7) to represent the day of the week: ";

std::cin >> dayNumber;

// Use a switch statement to determine the day of the week

switch (dayNumber) {

case 1:

std::cout << "Monday" << std::endl;

break;

case 2:

std::cout << "Tuesday" << std::endl;

break;

case 3:

std::cout << "Wednesday" << std::endl;

break;

case 4:

std::cout << "Thursday" << std::endl;

break;

case 5:

std::cout << "Friday" << std::endl;

break;

case 6:

std::cout << "Saturday" << std::endl;

break;

case 7:

std::cout << "Sunday" << std::endl;

break;

default:

std::cout << "Invalid input. Please enter a number between 1 and 7." << std::endl;

}

return 0;

}

### For loop

#include <iostream>

int main() {

// Declare variables

int limit, sum = 0;

// Prompt the user to enter a limit

std::cout << "Enter a positive integer as the limit: ";

std::cin >> limit;

// Check if the entered limit is positive

if (limit <= 0) {

std::cout << "Please enter a positive integer." << std::endl;

return 1; // Exit the program with an error code

}

// Use a for loop to calculate the sum of numbers from 1 to the limit

for (int i = 1; i <= limit; ++i) {

sum += i; // Add the current number to the sum

}

// Display the result

std::cout << "The sum of numbers from 1 to " << limit << " is: " << sum << std::endl;

return 0;

}

### While loop

#include <iostream>

int main() {

// Declare variables

int number, factorial = 1;

// Prompt the user to enter a positive integer

std::cout << "Enter a positive integer: ";

std::cin >> number;

// Check if the entered number is positive

if (number < 0) {

std::cout << "Please enter a positive integer." << std::endl;

return 1; // Exit the program with an error code

}

// Use a while loop to calculate the factorial of the number

int i = 1;

while (i <= number) {

factorial \*= i; // Multiply the current number to the factorial

++i; // Increment the loop variable

}

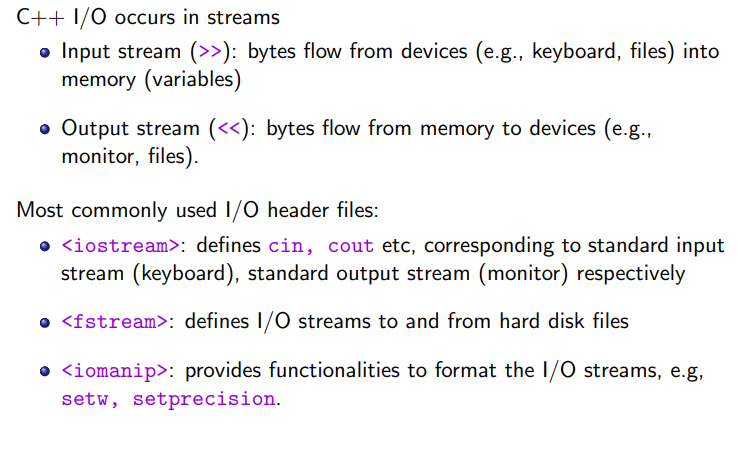
// Display the result

std::cout << "The factorial of " << number << " is: " << factorial << std::endl;

return 0;

}

### Input and Output



### C++ I/O Example to Console

#include <iostream>

#include <string>

int main() {

// Output: Displaying text to the console

std::cout << "Enter your name: ";

// Input: Reading user input using std::cin

std::string name;

std::getline(std::cin, name);

// Output: Displaying the user's name

std::cout << "Hello, " << name << "!" << std::endl;

// Input: Reading numeric input

std::cout << "Enter your age: ";

int age;

std::cin >> age;

// Output: Displaying the user's age

std::cout << "You are " << age << " years old." << std::endl;

return 0;

}

### C++ I/O example to File

#include <iostream>

#include <fstream>

#include <string>

int main() {

// Writing to a file

std::ofstream outputFile("output.txt");

if (outputFile.is\_open()) {

std::cout << "Enter a line to write to the file (type 'exit' to stop):" << std::endl;

std::string inputLine;

while (true) {

std::getline(std::cin, inputLine);

if (inputLine == "exit")

break;

outputFile << inputLine << std::endl;

}

std::cout << "Data written to the file 'output.txt'." << std::endl;

// Close the file

outputFile.close();

} else {

std::cerr << "Error opening the file for writing." << std::endl;

return 1;

}

// Reading from a file

std::ifstream inputFile("output.txt");

if (inputFile.is\_open()) {

std::cout << "Contents of the file 'output.txt':" << std::endl;

std::string line;

while (std::getline(inputFile, line)) {

std::cout << line << std::endl;

}

// Close the file

inputFile.close();

} else {

std::cerr << "Error opening the file for reading." << std::endl;

return 1;

}

return 0;

}

### Exception handling

In C++, you can use the throw statement to manually throw an exception. Exceptions are a way of handling errors or exceptional situations in your program. Here's a simple example:

#include <iostream>

// Function that throws an exception

void divideNumbers(int numerator, int denominator) {

if (denominator == 0) {

// Throw an exception if the denominator is zero

throw std::runtime\_error("Error: Division by zero");

}

// Perform the division if the denominator is not zero

int result = numerator / denominator;

std::cout << "Result of division: " << result << std::endl;

}

int main() {

try {

// Call a function that may throw an exception

divideNumbers(10, 2);

// Uncomment the line below to test the exception case

// divideNumbers(5, 0);

} catch (const std::exception& e) {

// Catch the exception and handle it

std::cerr << "Exception caught: " << e.what() << std::endl;

}

return 0;

}

In this example:

* The divideNumbers function takes two integers as parameters and checks if the denominator is zero. If it is, it throws a std::runtime\_error exception with an error message.
* In the main function, we have a try block where we call divideNumbers(10, 2); which should not throw an exception, and then we have a commented line // divideNumbers(5, 0); which, if uncommented, would result in an exception due to division by zero.
* The catch block catches any exceptions that are thrown within the try block. It prints an error message using std::cerr.
* Note: It's a good practice to catch exceptions by reference (const std::exception& e) to ensure proper handling and avoid object slicing.
* This is a basic example, and in real-world scenarios, you might create your own exception classes for more specific error handling.

## Quiz

1. Question: What is the size of the int data type in C++ on a 32-bit system?

a) 2 bytes

b) 4 bytes

c) 8 bytes

d) It depends on the compiler

1. Question: Which data type is used to store single-precision floating-point numbers in C++?

a) float

b) double

c) long double

d) float32

1. Question: In C++, what is the maximum value that can be stored in an unsigned short?

a) 32767

b) 65535

c) 2147483647

d) 4294967295

1. Question: How do you declare a character constant in C++?

a) char constant = 'A';

b) const char character = 'A';

c) constant character = 'A';

d) character constant := 'A';

1. Question: Which data type is used to represent true or false values in C++?

a) bool

b) int

c) char

d) double

1. Question: What is the difference between signed and unsigned data types in C++?

a) signed is used for integers, and unsigned is used for floating-point numbers

b) signed can represent both positive and negative values, while unsigned represents only positive values

c) There is no difference; the terms are interchangeable

d) unsigned is used for integers, and signed is used for floating-point numbers

1. Question: In C++, how do you initialize a variable at the time of declaration?

a) initialize int x = 10;

b) int x; x = 10;

c) int x(10);

d) int x = 10;

1. Question: What is the scope of a local variable in C++?

a) It is accessible throughout the entire program

b) It is accessible only within the function or block where it is declared

c) It is accessible in any function within the same file

d) It is accessible in any function within the program

1. Question: What is the correct syntax to declare a pointer variable in C++?

a) int \*ptr;

b) pointer int ptr;

c) ptr \* int;

d) declare pointer int ptr;