C++ and Object-Oriented Programming Study Notes

Topic 1: Introduction to Object-Oriented Programming

What is Object-Oriented Programming (OOP)?

Object-Oriented Programming is a programming paradigm that organizes code around objects and classes rather than functions and procedures. It's based on the concept of "objects" which contain data (attributes) and code (methods).

Key Features of OOP

1. **Encapsulation** - Bundling data and methods together in a single unit (class) and controlling access through access modifiers
2. **Inheritance** - Creating new classes based on existing classes, allowing code reuse and establishing parent-child relationships
3. **Polymorphism** - Ability to use the same interface for different data types or classes, enabling "one interface, multiple implementations"
4. **Abstraction** - Hiding complex implementation details and showing only essential features to the user

Uses and Importance of OOP

Uses:

- GUI Applications (Windows, buttons, menus as objects)
- Game Development (characters, weapons, items as objects)
- Web Development (user accounts, products, orders as objects)
- Database Management Systems
- Operating Systems
- Simulation Software

Importance:

- **Code Reusability**: Write once, use multiple times
- **Modularity**: Easy to maintain and modify
- **Scalability**: Easy to add new features
- **Security**: Data hiding and encapsulation
- **Problem Solving**: Models real-world entities naturally

Topic 2: Procedural vs Object-Oriented Programming

Procedural Programming Paradigm

Characteristics:

- Function-based approach
- Top-down approach

- Data and functions are separate
- Global data can be accessed by any function
- Focus on "what needs to be done"

Structure:

```
main()
 ├── function1()
 ├── function2()
 └── function3()
```

Advantages:

- Simple and easy to understand
- Suitable for small programs
- Faster execution for simple programs
- Easy debugging

Disadvantages:

- Difficult to maintain large programs
- Data security issues (global data)
- Code reusability is limited
- Difficult to model real-world problems

Object-Oriented Programming Paradigm

Characteristics:

- Object and class-based approach
- Bottom-up approach
- Data and functions are encapsulated together
- Data hiding through access modifiers
- Focus on "who is doing what"

Structure:

```
Class1 {
   data members
   member functions
}
Class2 inherits Class1 {
   additional members
}
```

Advantages:

- Better code organization
- Data security through encapsulation

- Code reusability through inheritance
- Easy to maintain and modify
- Models real-world problems effectively
- Polymorphism provides flexibility

Disadvantages:

- Steeper learning curve
- Slower execution compared to procedural
- More memory usage
- Over-engineering for simple problems

Comparison Table:

Feature	Procedural	Object-Oriented
Approach	Top-down	Bottom-up
Focus	Functions	Objects
Data Security	Low	High
Code Reusability	Limited	High
Maintenance	Difficult	Easy
Problem Solving	Step-by-step	Object interaction
Examples	C, Pascal	C++, Java, Python

Topic 3: Setting Up C++ Environment

Installing C++ Compiler

Windows:

1. **MinGW-w64**: Download from mingw-w64.org
2. **Visual Studio**: Download Community edition
3. **Code::Blocks**: Includes compiler
4. **Dev-C++**: Simple IDE with compiler

Linux/Ubuntu:

```bash
sudo apt update
sudo apt install g++
sudo apt install build-essential
```

macOS:

```bash
```

```
# Install Xcode Command Line Tools
xcode-select --install

# Or use Homebrew
brew install gcc
```

Code Editors and IDEs

Popular IDEs:

1. **Visual Studio Code**
   - Free, lightweight
   - Excellent C++ extension
   - Cross-platform
2. **Code::Blocks**
   - Free, open-source
   - Built-in compiler
   - Good for beginners
3. **Dev-C++**
   - Simple interface
   - Good for learning
   - Windows only
4. **CLion**
   - Professional IDE
   - Advanced debugging
   - Paid software
5. **Visual Studio**
   - Microsoft's IDE
   - Excellent IntelliSense
   - Windows focused

Essential Extensions for VS Code:

- C/C++ Extension Pack
- Code Runner
- C++ Intellisense
- Bracket Pair Colorizer

Basic Setup Verification

```
cpp
```

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

int main() {
    cout << "Hello, World!" << endl;
    return 0;
}
```

Compilation Commands:

```bash
g++ -o program program.cpp
./program
```

Topic 4: C++ Programming Statements, Variables, Data Types, and Scopes

C++ Program Structure

```cpp
#include <iostream>        // Preprocessor directive
using namespace std;        // Using directive

int main() {              // Main function
    // Program statements
    return 0;             // Return statement
}
```

Variables

Variables are named storage locations in memory that hold data values. They allow programs to store, modify, and retrieve information during execution.

Variable Declaration:

```cpp
datatype variable_name;
datatype variable_name = value;
```

Variable Naming Rules:

- Must start with letter or underscore
- Can contain letters, digits, underscores
- Case sensitive
- Cannot be C++ keywords
- Should be descriptive

Examples:

```cpp
```

```
int age;
double salary = 50000.0;
char grade = 'A';
string name = "John";
```

Data Types

Data types specify the type of data that a variable can store and determine the amount of memory allocated to the variable. They also define the operations that can be performed on the data.

1. Basic Data Types:

```cpp
// Integer types
int num = 10;          // 4 bytes
short shortNum = 5;    // 2 bytes
long longNum = 100000L; // 4 or 8 bytes
long long veryLong = 1000000LL; // 8 bytes

// Floating-point types
float pi = 3.14f;      // 4 bytes
double precise = 3.14159; // 8 bytes
long double morePrecise = 3.14159L; // 12 or 16 bytes

// Character type
char letter = 'A';     // 1 byte
char newline = '\n';   // Escape sequence

// Boolean type
bool isTrue = true;    // 1 byte
bool isFalse = false;
```

2. Derived Data Types:

```cpp
// Arrays
int numbers[5] = {1, 2, 3, 4, 5};
char word[10] = "Hello";

// Pointers
int* ptr = &num;
char* charPtr = &letter;

// References
int& ref = num;
```

3. User-Defined Data Types:

```cpp
```

```
// Structures
struct Student {
    int id;
    string name;
    float gpa;
};

// Classes
class Rectangle {
    int width, height;
public:
    int area() { return width * height; }
};

// Enumerations
enum Color {RED, GREEN, BLUE};
```

Variable Scopes

Variable scope determines the region of the program where a variable can be accessed. It defines the visibility and lifetime of variables in different parts of the code.

1. Global Scope:

```cpp
int globalVar = 20;    // Accessible throughout program

int main() {
    cout << globalVar;  // Can access global variable
    return 0;
}
```

2. Block Scope (includes function scope):

```cpp
int main() {                // Function block
    int x = 5;              // Function/block scope
    {
        int y = 10;         // Nested block scope
        cout << x;          // Can access outer block
    }
    // cout << y;           // Error: y not accessible here

    for (int i = 0; i < 5; i++) { // Loop block
        int z = i;          // Block scope within loop
    }
    // cout << z;           // Error: z not accessible here
    return 0;
}
```

3. Class Scope:

```cpp
class MyClass {
    int memberVar;      // Class scope - accessible to all member functions
public:
    void function1() {
        memberVar = 10; // Can access class member
    }
    void function2() {
        memberVar = 20; // Can also access same class member
    }
};
```

Constants

Constants are values that cannot be changed during program execution. They help prevent accidental modification of important values and make code more readable and maintainable.

```cpp
const int MAX_SIZE = 100;       // Constant variable
#define PI 3.14159              // Macro constant
const double E = 2.718;         // Constant expression
```

Topic 5: Basic Operations and Control Flow

Arithmetic Operators

Arithmetic operators are used to perform mathematical calculations on numeric data. They follow standard mathematical rules and operator precedence.

```cpp
```

```
int a = 10, b = 3;

// Basic arithmetic
int sum = a + b;       // Addition: 13
int diff = a - b;      // Subtraction: 7
int product = a * b;   // Multiplication: 30
int quotient = a / b;  // Division: 3 (integer division)
int remainder = a % b; // Modulus: 1

// Increment/Decrement
int x = 5;
x++;   // Post-increment: x becomes 6
++x;   // Pre-increment: x becomes 7
x--;   // Post-decrement: x becomes 6
--x;   // Pre-decrement: x becomes 5

// Compound assignment
x += 5; // x = x + 5
x -= 3; // x = x - 3
x *= 2; // x = x * 2
x /= 4; // x = x / 4
x %= 3; // x = x % 3
```

Logical Operators

Logical operators are used to combine or modify boolean expressions. They return true or false based on the logical relationship between operands and support short-circuit evaluation for efficiency.

```cpp
bool p = true, q = false;

// Logical AND
bool and_result = p && q;    // false
bool and_result2 = p && true; // true

// Logical OR
bool or_result = p || q;     // true
bool or_result2 = false || q; // false

// Logical NOT
bool not_result = !p;       // false
bool not_result2 = !q;      // true

// Short-circuit evaluation
int x = 5;
if (x > 0 && x < 10) {      // Both conditions checked
    cout << "x is between 0 and 10";
}
```

Bitwise Operators

Bitwise operators work at the bit level and perform operations on individual bits of integer data. They are commonly used in system programming, optimization, and when working with flags or masks.

```cpp
int a = 5;  // Binary: 0101
int b = 3;  // Binary: 0011

// Bitwise AND
int and_op = a & b;    // 0001 = 1

// Bitwise OR
int or_op = a | b;     // 0111 = 7

// Bitwise XOR
int xor_op = a ^ b;    // 0110 = 6

// Bitwise NOT
int not_op = ~a;       // Inverts all bits

// Left shift
int left_shift = a << 1; // 1010 = 10

// Right shift
int right_shift = a >> 1; // 0010 = 2
```

Control Flow Statements

Control flow statements determine the order in which program statements are executed. They allow programs to make decisions, repeat actions, and branch to different code sections based on conditions.

Conditional Statements:

Conditional statements execute different code blocks based on whether specified conditions are true or false.

1. if Statement:

```cpp
int score = 85;

if (score >= 90) {
    cout << "A grade";
} else if (score >= 80) {
    cout << "B grade";
} else if (score >= 70) {
    cout << "C grade";
} else {
    cout << "Below C grade";
}
```

2. switch Statement:

```cpp
```

```cpp
char grade = 'B';

switch (grade) {
    case 'A':
        cout << "Excellent!";
        break;
    case 'B':
        cout << "Good!";
        break;
    case 'C':
        cout << "Average";
        break;
    default:
        cout << "Invalid grade";
}
```

3. Ternary Operator:

```cpp
int a = 5, b = 3;
int max = (a > b) ? a : b;  // max = 5
```

Loop Statements:

Loop statements allow repeated execution of code blocks until a specified condition becomes false. They help avoid code repetition and process collections of data efficiently.

1. for Loop:

```cpp
// Basic for loop
for (int i = 0; i < 5; i++) {
    cout << i << " ";
}

// Range-based for loop (C++11)
int arr[] = {1, 2, 3, 4, 5};
for (int element : arr) {
    cout << element << " ";
}
```

2. while Loop:

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

3. do-while Loop:

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

Loop Control Statements:

```cpp
for (int i = 0; i < 10; i++) {
    if (i == 3) {
        continue;  // Skip iteration when i = 3
    }
    if (i == 7) {
        break;     // Exit loop when i = 7
    }
    cout << i << " ";
}
```

Topic 6: C++ Modular Programming using Functions

Function Basics

Functions are reusable blocks of code that perform specific tasks. They promote code modularity, reusability, and make programs easier to understand, test, and maintain by breaking complex problems into smaller, manageable pieces.

Function Declaration (Prototype):

```cpp
return_type function_name(parameter_list);

// Examples
int add(int a, int b);
void displayMessage();
double calculateArea(double radius);
```

Function Definition:

```cpp
return_type function_name(parameter_list) {
    // Function body
    return value; // if return type is not void
}
```

Function Call:

```cpp
```

```
int result = add(5, 3);
displayMessage();
double area = calculateArea(5.0);
```

Complete Function Example

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

// Function declaration
int add(int a, int b);
void printResult(int result);

int main() {
    int x = 10, y = 20;
    int sum = add(x, y);       // Function call
    printResult(sum);          // Function call
    return 0;
}

// Function definition
int add(int a, int b) {
    return a + b;
}

void printResult(int result) {
    cout << "The result is: " << result << endl;
}
```

Types of Functions

Functions can be categorized based on their source and implementation. Understanding different types helps in choosing the right function for specific programming needs.

1. Built-in Functions:

These are predefined functions provided by C++ libraries that perform common tasks like mathematical calculations, string manipulations, and input/output operations.

```cpp
#include <cmath>
#include <cstring>

// Math functions
double result = sqrt(16);      // Square root
double power = pow(2, 3);      // Power function
double sine = sin(3.14159/2);  // Sine function

// String functions
char str[] = "Hello";
int length = strlen(str);      // String length
```

2. User-defined Functions:

These are custom functions created by programmers to perform specific tasks according to program requirements. They allow code customization and promote code organization.

```cpp
// Function with no parameters
void greet() {
    cout << "Hello, World!" << endl;
}

// Function with parameters
int multiply(int a, int b) {
    return a * b;
}

// Function with default parameters
int power(int base, int exponent = 2) {
    int result = 1;
    for (int i = 0; i < exponent; i++) {
        result *= base;
    }
    return result;
}
```

Recursive Functions

Recursive functions are functions that call themselves to solve problems by breaking them down into smaller, similar subproblems. They consist of a base case (stopping condition) and a recursive case (function calls itself).

```cpp
// Factorial using recursion
int factorial(int n) {
    if (n <= 1) {
        return 1;  // Base case
    }
    return n * factorial(n - 1);  // Recursive case
}

// Fibonacci sequence
int fibonacci(int n) {
    if (n <= 1) {
        return n;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Best Practices for Functions

Following good practices when writing functions leads to cleaner, more maintainable, and more efficient code.

1. **Use meaningful names**: `calculateArea()` instead of `calc()`
2. **Keep functions small**: One function, one purpose
3. **Use const for parameters that shouldn't change**: `void print(const string& message)`
4. **Return early when possible**: Avoid deeply nested conditions
5. **Use default parameters wisely**: `void display(int value, bool showLabel = true)`

Practice Problems

Set 1: Basic Concepts and Syntax

Easy Problems:

1. **Variable Declaration and Initialization**: Write a C++ program that declares variables of different data types (int, float, char, bool) and initializes them with appropriate values. Display all values.
2. **Simple Arithmetic**: Create a program that takes two numbers as input and performs all arithmetic operations (+, -, *, /, %) on them. Display the results.
3. **Basic Input/Output**: Write a C++ program that takes a person's name, age, and salary as input, then displays them in a formatted manner with appropriate labels.

Medium Problems:

4. **Scope Demonstration**: Write a C++ program that demonstrates global scope, block scope, and class scope by declaring variables in different locations and showing where they can be accessed.
5. **Data Type Size Calculator**: Create a program that displays the size (in bytes) of all basic data types in C++ using the sizeof operator. Also calculate and display how many variables of each type can fit in 1KB of memory.

Hard Problem:

6. **Variable Scope Challenge**: Write a C++ program with nested blocks and functions that demonstrates variable shadowing. Create variables with the same name in different scopes and show how scope resolution works.

Set 2: Operators and Control Flow

Easy Problems:

1. **Basic Operators**: Write a program that demonstrates the use of logical operators (&&, ||, !) with boolean variables. Show truth tables for each operator.
2. **Simple Loop**: Create a program using a for loop to print numbers from 1 to 10, and then print even numbers from 2 to 20 using a while loop.
3. **If-Else Chain**: Write a program that takes a student's marks as input and displays the grade using if-else statements (A: 90-100, B: 80-89, C: 70-79, F: below 70).

Medium Problems:

4. **Bitwise Operations**: Write a program that takes two integers and performs all bitwise operations (&, |, ^, ~, <<, >>) on them. Display results in both decimal and binary format.
5. **Nested Loops Pattern**: Create a program that uses nested loops to print the following pattern:

```
*
**
***
****
*****
```

Hard Problem:

6. **Complex Control Flow**: Write a program that implements a simple calculator using switch statements. It should handle multiple operations in sequence and use loops to continue until the user chooses to exit. Include error handling for invalid inputs.

Set 3: Functions and Advanced Concepts

Easy Problems:

1. **Basic Function**: Write a function that takes two integers as parameters and returns their sum. Call this function from main() and display the result.
2. **Simple Function Types**: Create a program showing examples of both built-in functions (like sqrt, pow) and user-defined functions. Use at least 3 built-in functions from different libraries.
3. **Function with No Return**: Write a void function that takes a string as parameter and displays it with a border made of asterisks.

Medium Problems:

4. **Multiple Function Calls**: Create a program with functions to calculate area of circle, rectangle, and triangle. Use a menu system to let users choose which area to calculate.
5. **Recursive Function**: Write a recursive function to calculate the factorial of a number. Also write an iterative version and compare both approaches in terms of logic.

Hard Problem:

6. **Comprehensive Program**: Design a program that combines all concepts learned. Create a simple student management system that uses functions to add student data (name, roll number, marks), calculate grades, and display student information. Use appropriate data types, control structures, and demonstrate proper scope usage.