Introduction to C++ - Practice Exercises

Worksheet for Students

Computer Science Department

2025-10-10

Instructions

This worksheet contains practice exercises covering the fundamental concepts of C++ programming. These exercises are designed for self-study and practice. Complete each exercise in your own development environment, test your code, and make sure it compiles without errors. Focus on using modern C++ practices (C++17/20).

Practice Guidelines:

- Use meaningful variable names
- Add comments to explain your logic
- Test your code with different inputs
- Follow modern C++ best practices
- Handle potential errors appropriately
- Experiment with different approaches

Section 1: Basic Types and Variables

Exercise 1.1: Variable Declaration and Initialization

Write a program that declares variables of different types using modern initialization syntax:

Requirements:

a) Declare an integer variable `age` and initialize it to 25 using uniform initialization
b) Declare a double variable `pi` using `constexpr` and set it to 3.14159
c) Declare a boolean variable `is_student` and set it to `true`
d) Declare a string variable `name` and initialize it to your name
e) Use `auto` to declare a variable that stores the result of `10 * 5`
f) Print all variables to the console

Expected Output:

```
Age: 25
Pi: 3.14159
Is Student: 1
Name: [Your Name]
Result: 50
```

Exercise 1.2: Type Sizes

Write a program that displays the size (in bytes) of the following types:

- `bool`, `char`, `short`, `int`, `long`, `long long`
- `float`, `double`
- Your system's pointer size using `void*`

Also, use `std::numeric_limits` to print the minimum and maximum values for `int` and `double`.

Section 2: Functions

Exercise 2.1: Simple Functions

Write the following functions and test them in `main()`:

a) `square(double x)` - returns the square of x
b) `is_even(int n)` - returns true if n is even, false otherwise
c) `max_of_three(int a, int b, int c)` - returns the largest of three integers
d) `print_separator()` - a void function that prints "`==========`" to console

Exercise 2.2: Function with Default Parameters

Create a function `greet(std::string name, std::string greeting = "Hello")` that prints a greeting message. If no greeting is provided, it should use "Hello" as default.

Test it with: - `greet("Alice")` - `greet("Bob", "Good morning")`

Section 3: Arrays and Loops

Exercise 3.1: Array Operations

Write a program that:

a) Creates a `std::array<int, 5>` with values {10, 20, 30, 40, 50}
b) Prints all elements using a traditional for loop
c) Prints all elements using a range-based for loop
d) Calculates and prints the sum of all elements
e) Finds and prints the maximum value

Exercise 3.2: Vector Manipulation

Write a program that:

a) Creates an empty `std::vector<int>`
b) Adds the numbers 1 through 10 to the vector using a loop

c) Doubles each element in the vector using a range-based for loop with references
d) Removes all elements greater than 15
e) Prints the final contents

Hint: Use `std::remove_if` with `std::erase` or a manual loop with iterators.

Section 4: Pointers and References

Exercise 4.1: Understanding References

Complete this program by implementing the missing functions:

```
#include <iostream>

void swap_by_value(int a, int b) {
    // TODO: Implement (this won't work for swapping)
}

void swap_by_reference(/* TODO: Add parameters */) {
    // TODO: Implement correct swap
}

int main() {
    int x = 10, y = 20;

    std::cout << "Before: x = " << x << ", y = " << y << "\n";

    swap_by_value(x, y);
    std::cout << "After swap_by_value: x = " << x << ", y = " << y << "\n";

    swap_by_reference(x, y);
    std::cout << "After swap_by_reference: x = " << x << ", y = " << y <<
"\n";

    return 0;
}
```

Expected Output:

```
Before: x = 10, y = 20
After swap_by_value: x = 10, y = 20
After swap_by_reference: x = 20, y = 10
```

Exercise 4.2: Pointer Basics

Write a program that:

a) Declares an integer variable `num` with value 42
b) Creates a pointer `ptr` that points to `num`

c) Prints the value of `num`, the address of `num`, and the value pointed by `ptr`
d) Changes the value of `num` through the pointer to 100
e) Verifies that `num` has changed

Section 5: Structures and Classes

Exercise 5.1: Student Structure

Create a `Student` struct with the following members: - `std::string name` - `int id` - `double gpa`

Then write a program that:

a) Creates three Student objects with different data
b) Stores them in a `std::vector<Student>`
c) Prints information for all students
d) Finds and prints the student with the highest GPA

Exercise 5.2: Rectangle Class

Create a `Rectangle` class with:

Private members: - `double width` - `double height`

Public members: - Constructor that takes width and height - `area()` method that returns the area - `perimeter()` method that returns the perimeter - `is_square()` method that returns true if width equals height - `scale(double factor)` method that multiplies both dimensions by factor

Test your class by creating rectangles and calling all methods.

Section 6: Enumerations

Exercise 6.1: Traffic Light System

Create an `enum class TrafficLight` with values: `red`, `yellow`, `green`.

Write a function `get_action(TrafficLight light)` that returns a string: - Red → "Stop" - Yellow → "Prepare to stop" - Green → "Go"

Also implement an `operator++` that cycles through the lights in order (green → yellow → red → green).

Example usage:

```
TrafficLight light = TrafficLight::green;
std::cout << get_action(light) << "\n";  // Output: Go
++light;
std::cout << get_action(light) << "\n";  // Output: Prepare to stop
```

Exercise 6.2: Days of Week

Create an `enum class Day` representing days of the week (Monday through Sunday).

Write functions: - `is_weekend(Day d)` - returns true for Saturday and Sunday - `day_name(Day d)` - returns the string name of the day - `next_day(Day d)` - returns the next day (Sunday wraps to Monday)

Section 7: Error Handling

Exercise 7.1: Safe Division

Write a function `safe_divide(double a, double b)` that:

- Returns `a / b` if `b` is not zero
- Throws `std::invalid_argument` exception if `b` is zero

Write a main function that uses try-catch to handle the exception properly.

Example:

```
try {
    double result = safe_divide(10.0, 2.0);
    std::cout << "Result: " << result << "\n";

    result = safe_divide(10.0, 0.0);  // Should throw
} catch (const std::invalid_argument& e) {
    std::cout << "Error: " << e.what() << "\n";
}
```

Exercise 7.2: Array Bounds Checking

Create a class `SafeArray` that:

- Has a private `std::array<int, 10>` member
- Has an `at(int index)` method that returns the element at index
- Throws `std::out_of_range` if index is invalid
- Has a `set(int index, int value)` method with the same error checking

Test your class with both valid and invalid indices.

Section 8: Templates and Generic Programming

Exercise 8.1: Generic Maximum Function

Write a template function `maximum` that works with any type that supports the > operator:

```
template<typename T>
T maximum(T a, T b) {
    // TODO: Implement
}
```

Test it with: - Integers: `maximum(5, 10)` - Doubles: `maximum(3.14, 2.71)` - Strings: `maximum(std::string("apple"), std::string("banana"))`

Exercise 8.2: Generic Container Statistics

Write a template function that calculates the average of elements in any container:

```
template<typename Container>
double average(const Container& c) {
    // TODO: Implement
    // Hint: use range-based for loop
}
```

Test it with: - `std::vector<int>` - `std::array<double, 5>` - `std::list<float>`

Hint: You'll need to include `<numeric>` or manually sum and divide.

Section 9: Modern C++ Features

Exercise 9.1: Smart Pointers

Rewrite this code to use `std::unique_ptr` instead of raw pointers:

```
#include <iostream>

class Resource {
public:
    Resource() { std::cout << "Resource acquired\n"; }
    ~Resource() { std::cout << "Resource destroyed\n"; }
    void use() { std::cout << "Resource in use\n"; }
};

int main() {
    Resource* ptr = new Resource();
    ptr->use();
    delete ptr;  // Easy to forget!
    return 0;
}
```

Your version should automatically manage memory and produce the same output.

Exercise 9.2: Lambda Expressions

Write a program that:

a) Creates a `std::vector<int>` with values {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
b) Uses `std::count_if` with a lambda to count even numbers
c) Uses `std::for_each` with a lambda to print each element
d) Uses `std::transform` with a lambda to square each element

Hint: Include `<algorithm>` for these functions.

Section 10: Comprehensive Challenge

Exercise 10.1: Bank Account System

Create a complete bank account system with the following requirements:

1. Create an `enum class AccountType`: - `checking`, `savings`, `business`

2. Create a `BankAccount` class with:

Private members: - `std::string owner_name` - `int account_number` - `double balance` - `AccountType type`

Public members: - Constructor that initializes all members - `deposit(double amount)` - throws exception if amount <= 0 - `withdraw(double amount)` - throws exception if insufficient funds or amount <= 0 - `get_balance()` const - returns current balance - `get_info()` const - returns formatted string with account info - `transfer(BankAccount& other, double amount)` - transfers money to another account

3. In main(): - Create at least 3 different accounts - Perform various operations (deposits, withdrawals, transfers) - Use try-catch blocks to handle exceptions - Print account information after each operation

Example Output:

```
Account: John Doe (#1001) - Checking
Balance: $1000.00

Depositing $500...
New Balance: $1500.00

Transferring $200 to Account #1002...
Transfer successful!
```

Section 11: Bonus Challenges

Exercise 11.1: Temperature Converter

Create a temperature conversion system:

1. Create an `enum class TempScale`: `celsius`, `fahrenheit`, `kelvin`

2. Create a `Temperature` class: - Stores value and scale - Has methods to convert to other scales - Overloads comparison operators (<, >, ==) - Overloads arithmetic operators (+, -)

3. Formulas: - C to F: `(C × 9/5) + 32` - C to K: `C + 273.15` - F to C: `(F - 32) × 5/9` - K to C: `K - 273.15`

Example usage:

```
Temperature t1{100.0, TempScale::celsius};
Temperature t2{212.0, TempScale::fahrenheit};

if (t1 == t2) {
    std::cout << "Same temperature!\n";
}
```

Exercise 11.2: Simple Vector Class

Implement a simplified version of `std::vector` called `SimpleVector`:

Requirements: - Uses dynamic memory allocation with smart pointers - Has `push_back()`, `pop_back()`, `size()`, `capacity()` methods - Implements `operator[]` for element access - Has proper copy/move constructors and assignment operators - Automatically resizes when capacity is reached (e.g., double capacity)

Bonus: Implement `begin()` and `end()` methods to support range-based for loops.

Tips for Practice

General Approach

1. **Start Small**: Begin with the simplest version of each exercise
2. **Test Frequently**: Compile and test after each small change
3. **Read Error Messages**: Compiler errors contain valuable information
4. **Use Debugger**: Step through your code to understand behavior
5. **Experiment**: Try different approaches and see what works

Modern C++ Best Practices

Use Uniform Initialization

```
// Preferred
int x{42};
std::vector<int> vec{1, 2, 3};

// Avoid narrowing conversions
int i{7.2};  // Compilation error - good!
```

Prefer const and constexpr

```
const int MAX_SIZE = 100;        // Runtime constant
constexpr double PI = 3.14159;   // Compile-time constant
```

Use auto When Appropriate

```
auto value = calculate_something();  // Clear from context
auto it = container.begin();         // Iterator types are verbose
```

Range-Based For Loops

```
// Read-only
for (const auto& item : container) {
    std::cout << item << "\n";
}

// Modify elements
for (auto& item : container) {
    item *= 2;
}
```

Common Pitfalls

Forgetting & in Range-Based Loops

```
// Wrong - creates copies
for (auto item : large_objects) { }

// Correct - uses references
for (auto& item : large_objects) { }
for (const auto& item : large_objects) { }  // Read-only
```

Not Checking Container Bounds

```
// Dangerous
int value = vec[100];  // No bounds checking

// Safe
int value = vec.at(100);  // Throws exception if out of bounds
```

Forgetting const on Methods

```
class Example {
    int value;
public:
    // Wrong - can't be called on const objects
    int get_value() { return value; }

    // Correct
    int get_value() const { return value; }
};
```

Additional Resources

Online References

- **C++ Reference:** cppreference.com

  - Comprehensive documentation for all C++ features
  - Examples and best practices
- **Compiler Explorer:** godbolt.org
  - See how your code compiles
  - Try different compilers and optimization levels
- **C++ Core Guidelines:** isocpp.github.io/CppCoreGuidelines
  - Best practices from C++ experts
  - Modern C++ idioms

Recommended Books

1. **"A Tour of C++"** - Bjarne Stroustrup
   - Quick overview of modern C++
   - Written by the creator of C++
2. **"Effective Modern C++"** - Scott Meyers
   - 42 specific ways to improve your C++11/14 code
   - Essential for modern C++ development
3. **"C++ Primer"** - Stanley Lippman
   - Comprehensive introduction
   - Covers fundamentals through advanced topics

Tools

Compilers

- **GCC** (Linux/Mac): `g++ -std=c++17 -Wall -Wextra program.cpp`
- **Clang** (Linux/Mac): `clang++ -std=c++17 -Wall -Wextra program.cpp`
- **MSVC** (Windows): Part of Visual Studio

IDEs

- Visual Studio Code (with C++ extensions)
- CLion (by JetBrains)
- Visual Studio (Windows)
- Code::Blocks (cross-platform)

Happy Coding!

Remember: The best way to learn programming is by doing. Don't just read the exercises—write the code, experiment, and make mistakes. That's how you learn!

Section 2: Functions

Exercise 2.1: Simple Functions

Write the following functions and test them in `main()`:

a) `square(double x)` - returns the square of x
b) `is_even(int n)` - returns true if n is even, false otherwise
c) `max_of_three(int a, int b, int c)` - returns the largest of three integers

d) `print_separator()` - a void function that prints "==========" to console

Exercise 2.2: Function with Default Parameters

Create a function `greet(std::string name, std::string greeting = "Hello")` that prints a greeting message. If no greeting is provided, it should use "Hello" as default.

Test it with: - `greet("Alice")` - `greet("Bob", "Good morning")`

Section 3: Arrays and Loops

Exercise 3.1: Array Operations

Write a program that:

a) Creates a `std::array<int, 5>` with values {10, 20, 30, 40, 50}
b) Prints all elements using a traditional for loop
c) Prints all elements using a range-based for loop
d) Calculates and prints the sum of all elements
e) Finds and prints the maximum value

Exercise 3.2: Vector Manipulation

Write a program that:

a) Creates an empty `std::vector<int>`
b) Adds the numbers 1 through 10 to the vector using a loop
c) Doubles each element in the vector using a range-based for loop with references
d) Removes all elements greater than 15
e) Prints the final contents

Section 4: Pointers and References

Exercise 4.1: Understanding References

Complete this program by filling in the missing parts:

```
#include <iostream>

void swap_by_value(int a, int b) {
    // TODO: Implement (this won't work for swapping)
}

void swap_by_reference(/* TODO: Add parameters */) {
    // TODO: Implement correct swap
}

int main() {
    int x = 10, y = 20;
```

```
    std::cout << "Before: x = " << x << ", y = " << y << "\n";

    swap_by_value(x, y);
    std::cout << "After swap_by_value: x = " << x << ", y = " << y << "\n";

    swap_by_reference(x, y);
    std::cout << "After swap_by_reference: x = " << x << ", y = " << y <<
"\n";

    return 0;
}
```

Exercise 4.2: Pointer Basics

Write a program that:

a) Declares an integer variable `num` with value 42
b) Creates a pointer `ptr` that points to `num`
c) Prints the value of `num`, the address of `num`, and the value pointed by `ptr`
d) Changes the value of `num` through the pointer to 100
e) Verifies that `num` has changed

Section 5: Structures and Classes

Exercise 5.1: Student Structure

Create a `Student` struct with the following members: - `std::string name` - `int id` - `double gpa`

Then write a program that:

a) Creates three Student objects with different data
b) Stores them in a `std::vector<Student>`
c) Prints information for all students
d) Finds and prints the student with the highest GPA

Exercise 5.2: Rectangle Class

Create a `Rectangle` class with:

Private members: - `double width` - `double height`

Public members: - Constructor that takes width and height - `area()` method that returns the area - `perimeter()` method that returns the perimeter - `is_square()` method that returns true if width equals height - `scale(double factor)` method that multiplies both dimensions by factor

Test your class by creating rectangles and calling all methods.

Section 6: Enumerations

Exercise 6.1: Traffic Light System

Create an `enum class TrafficLight` with values: `red`, `yellow`, `green`.

Write a function `get_action(TrafficLight light)` that returns a string: - Red → "Stop" - Yellow → "Prepare to stop" - Green → "Go"

Also implement an `operator++` that cycles through the lights in order.

Exercise 6.2: Days of Week

Create an `enum class Day` representing days of the week.

Write functions: - `is_weekend(Day d)` - returns true for Saturday and Sunday - `day_name(Day d)` - returns the string name of the day - `next_day(Day d)` - returns the next day (Sunday follows Saturday)

Section 7: Error Handling

Exercise 7.1: Safe Division

Write a function `safe_divide(double a, double b)` that:

- Returns `a / b` if `b` is not zero
- Throws `std::invalid_argument` exception if `b` is zero

Write a main function that uses try-catch to handle the exception properly.

Exercise 7.2: Array Bounds Checking

Create a class `SafeArray` that:

- Has a private `std::array<int, 10>` member
- Has an `at(int index)` method that returns the element at index
- Throws `std::out_of_range` if index is invalid
- Has a `set(int index, int value)` method with the same error checking

Test your class with both valid and invalid indices.

Section 8: Templates and Generic Programming

Exercise 8.1: Generic Maximum Function

Write a template function `maximum` that works with any type that supports the > operator:

```
template<typename T>
T maximum(T a, T b) {
    // TODO: Implement
}
```

Test it with: - Integers: `maximum(5, 10)` - Doubles: `maximum(3.14, 2.71)` - Strings: `maximum(std::string("apple"), std::string("banana"))`

Exercise 8.2: Generic Container Statistics

Write a template function that calculates the average of elements in any container:

```
template<typename Container>
double average(const Container& c) {
    // TODO: Implement
    // Hint: use range-based for loop
}
```

Test it with: - `std::vector<int>` - `std::array<double, 5>` - `std::list<float>`

Section 9: Modern C++ Features

Exercise 9.1: Smart Pointers

Rewrite this code to use `std::unique_ptr` instead of raw pointers:

```
#include <iostream>

class Resource {
public:
    Resource() { std::cout << "Resource acquired\n"; }
    ~Resource() { std::cout << "Resource destroyed\n"; }
    void use() { std::cout << "Resource in use\n"; }
};

int main() {
    Resource* ptr = new Resource();
    ptr->use();
    delete ptr;  // Easy to forget!
    return 0;
}
```

Exercise 9.2: Lambda Expressions

Write a program that:

a) Creates a `std::vector<int>` with values {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
b) Uses `std::count_if` with a lambda to count even numbers
c) Uses `std::for_each` with a lambda to print each element
d) Uses `std::transform` with a lambda to square each element

Section 10: Comprehensive Challenge

Exercise 10.1: Bank Account System

Create a complete bank account system with the following requirements:

1. Create an `enum class AccountType`: - `checking`, `savings`, `business`

2. Create a `BankAccount` class with:

Private members: - `std::string owner_name` - `int account_number` - `double balance` - `AccountType type`

Public members: - Constructor that initializes all members - `deposit(double amount)` - throws exception if amount <= 0 - `withdraw(double amount)` - throws exception if insufficient funds or amount <= 0 - `get_balance()` const - returns current balance - `get_info()` const - returns formatted string with account info - `transfer(BankAccount& other, double amount)` - transfers money to another account

3. In main(): - Create at least 3 different accounts - Perform various operations (deposits, withdrawals, transfers) - Use try-catch blocks to handle exceptions - Print account information after each operation

Section 11: Bonus Challenges

Exercise 11.1: Temperature Converter

Create a temperature conversion system:

1. Create an `enum class TempScale`: `celsius`, `fahrenheit`, `kelvin`

2. Create a `Temperature` class: - Stores value and scale - Has methods to convert to other scales - Overloads comparison operators (<, >, ==) - Overloads arithmetic operators (+, -)

3. Formulas: - C to F: `(C × 9/5) + 32` - C to K: `C + 273.15` - F to C: `(F - 32) × 5/9` - K to C: `K - 273.15`

Exercise 11.2: Simple Vector Class

Implement a simplified version of `std::vector` called `SimpleVector`:

Requirements: - Uses dynamic memory allocation with smart pointers - Has `push_back()`, `pop_back()`, `size()`, `capacity()` methods - Implements `operator[]` for element access - Has proper copy/move constructors and assignment operators - Automatically resizes when capacity is reached

Additional Resources

- **C++ Reference:** cppreference.com
- **Compiler Explorer:** godbolt.org
- **C++ Core Guidelines:** isocpp.github.io/CppCoreGuidelines

Good luck with your exercises!