Functions in C++

CSI-140 Introduction to Programming

Instructor: Dr. Vikas Thammanna Gowda Semester: Fall 2025

1 Introduction to Functions

In programming, a **function** is a self-contained block of code designed to perform a specific task. Think of a function as a small machine that takes some input (optional), processes it, and produces some output (optional). Functions are one of the most important concepts in programming because they allow us to break down complex problems into smaller, manageable pieces.

Imagine you're baking cookies multiple times. Instead of writing out the entire recipe each time, you could write it once and simply refer to it whenever needed. Functions work the same way in programming - you write the code once and can use it many times throughout your program.

Functions provide several key benefits:

- **Code Reusability:** Write once, use many times
- **Modularity:** Break complex problems into smaller, manageable parts
- **Easier Debugging:** Test and fix problems in isolated pieces of code
- **Better Organization:** Group related code together
- **Abstraction:** Hide complex implementation details behind simple interfaces

C++ provides two categories of functions:

1. **Built-in Functions:** Pre-written functions provided by C++ through its standard library
2. **User-Defined Functions:** Functions that programmers create for their specific needs

2 Built-in Functions

2.1 What are Built-in Functions?

Built-in functions (also called library functions or standard functions) are pre-written functions that come with C++. These functions are tested, optimized, and ready to use. They handle common tasks like mathematical calculations, input/output operations, string manipulation, and more. To use these functions, you typically need to include the appropriate header file.

Think of built-in functions as tools in a toolbox that C++ provides. Just as you don't need to manufacture a hammer to use one, you don't need to write these functions yourself - you simply need to know they exist and how to use them.

2.2 Mathematical Functions (cmath header)

The `<cmath>` header provides a rich set of mathematical functions. These functions handle complex mathematical operations that would be tedious or error-prone to implement yourself.

2.2.1 Common Mathematical Functions

Example: Basic mathematical functions

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

int main() {
    double number = 16.0;
    double decimal_num = 4.7;
    double negative_num = -15.3;

    // Square root function
    cout << "Square root of " << number << " = " << sqrt(number) <<
        endl;

    // Power function: pow(base, exponent)
    cout << "2 raised to power 3 = " << pow(2, 3) << endl;
    cout << "5 raised to power 2 = " << pow(5, 2) << endl;

    // Absolute value functions
    cout << "Absolute value of " << negative_num << " = " << abs(
        negative_num) << endl;
    cout << "Absolute value (floating) of " << negative_num << " = "
        << fabs(negative_num) << endl;

    // Ceiling and floor functions
    cout << "Ceiling of " << decimal_num << " = " << ceil(decimal_num
        ) << endl;  // Rounds up
    cout << "Floor of " << decimal_num << " = " << floor(decimal_num)
         << endl;   // Rounds down

    // Round function
    cout << "Round of " << decimal_num << " = " << round(decimal_num)
         << endl;   // Normal rounding
    cout << "Round of 4.5 = " << round(4.5) << endl;   // Rounds to
        nearest even

    return 0;
}
```

Output:

```
Square root of 16 = 4
2 raised to power 3 = 8
5 raised to power 2 = 25
Absolute value of -15.3 = 15
Absolute value (floating) of -15.3 = 15.3
Ceiling of 4.7 = 5
Floor of 4.7 = 4
Round of 4.7 = 5
Round of 4.5 = 4
```

2.2.2 Logarithmic and Exponential Functions

Example: Logarithmic and exponential functions

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

int main() {
    double value = 10.0;

    // Exponential function
    cout << "e^2 = " << exp(2) << endl;
    cout << "e^1 = " << exp(1) << endl;

    // Natural logarithm (base e)
    cout << "Natural log of " << value << " = " << log(value) << endl
        ;
    cout << "Natural log of e = " << log(exp(1)) << endl;

    // Common logarithm (base 10)
    cout << "Log base 10 of " << value << " = " << log10(value) <<
        endl;
    cout << "Log base 10 of 100 = " << log10(100) << endl;

    // Base 2 logarithm
    cout << "Log base 2 of 8 = " << log2(8) << endl;
    cout << "Log base 2 of 16 = " << log2(16) << endl;

    return 0;
}
```

Output:

```
e^2 = 7.38906
e^1 = 2.71828
Natural log of 10 = 2.30259
Natural log of e = 1
Log base 10 of 10 = 1
Log base 10 of 100 = 2
Log base 2 of 8 = 3
Log base 2 of 16 = 4
```

2.3 Character Functions (cctype header)

The `<cctype>` header provides functions for character classification and conversion. These functions are essential when processing text input.

Example: Character classification functions

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

int main() {
    char letter = 'A';
    char digit = '5';
    char space = ' ';
    char symbol = '@';
    char lower_case = 'g';

    // Character classification
    cout << "Is '" << letter << "' alphabetic? " << (isalpha(letter)
       ? "Yes" : "No") << endl;
    cout << "Is '" << digit << "' a digit? " << (isdigit(digit) ? "
       Yes" : "No") << endl;
    cout << "Is '" << space << "' a space? " << (isspace(space) ? "
       Yes" : "No") << endl;
    cout << "Is '" << letter << "' uppercase? " << (isupper(letter) ?
        "Yes" : "No") << endl;
    cout << "Is '" << lower_case << "' lowercase? " << (islower(
       lower_case) ? "Yes" : "No") << endl;
    cout << "Is '" << digit << "' alphanumeric? " << (isalnum(digit)
       ? "Yes" : "No") << endl;
    cout << "Is '" << symbol << "' punctuation? " << (ispunct(symbol)
        ? "Yes" : "No") << endl;

    cout << "\nCharacter conversion:" << endl;

    // Character conversion
    cout << "Uppercase of '" << lower_case << "' = '" << (char)
       toupper(lower_case) << "'" << endl;
    cout << "Lowercase of '" << letter << "' = '" << (char)tolower(
       letter) << "'" << endl;

    // Converting a string
    string text = "Hello World 123!";
    cout << "\nOriginal text: " << text << endl;
    cout << "Converted to uppercase: ";
    for (int i = 0; i < text.length(); i++) {
        cout << (char)toupper(text[i]);
    }
    cout << endl;

    return 0;
}
```

Output:

```
Is 'A' alphabetic? Yes
Is '5' a digit? Yes
Is ' ' a space? Yes
Is 'A' uppercase? Yes
Is 'g' lowercase? Yes
Is '5' alphanumeric? Yes
Is '@' punctuation? Yes

Character conversion:
Uppercase of 'g' = 'G'
Lowercase of 'A' = 'a'

Original text: Hello World 123!
Converted to uppercase: HELLO WORLD 123!
```

2.4 String Functions (string header)

The `<string>` header provides the string class with many useful member functions. While these are technically member functions of the string class rather than standalone functions, they're built-in and essential for string manipulation.

Example: String manipulation functions

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

int main() {
    string text = "Hello World";
    string name = "Alice";
    string empty_str = "";

    // Length and size functions
    cout << "Length of '" << text << "' = " << text.length() << endl;
    cout << "Size of '" << text << "' = " << text.size() << endl;  //
        Same as length()

    // Empty check
    cout << "Is '" << empty_str << "' empty? " << (empty_str.empty()
       ? "Yes" : "No") << endl;
    cout << "Is '" << text << "' empty? " << (text.empty() ? "Yes" :
       "No") << endl;

    // String concatenation
    string greeting = "Hello, " + name + "!";
    cout << "Concatenation: " << greeting << endl;

    // Append function
    string message = "Good ";
    message.append("morning");
    cout << "After append: " << message << endl;

    // Substring function
    string full_text = "Programming in C++";
    string sub = full_text.substr(0, 11);  // Starting at index 0,
       take 11 characters
    cout << "Substring: " << sub << endl;

    // Find function
    string sentence = "The quick brown fox";
    size_t position = sentence.find("quick");
    if (position != string::npos) {
        cout << "'quick' found at position: " << position << endl;
    }

    // Character access
    cout << "First character of '" << text << "': " << text[0] <<
       endl;
    cout << "Last character of '" << text << "': " << text[text.
       length() - 1] << endl;

    // Clear function
    string temp = "Temporary";
    cout << "Before clear: " << temp << endl;
    temp.clear();
    cout << "After clear: '" << temp << "' (empty)" << endl;

    return 0;
}
```

Output:

```
Length of 'Hello World' = 11
Size of 'Hello World' = 11
Is '' empty? Yes
Is 'Hello World' empty? No
Concatenation: Hello, Alice!
After append: Good morning
Substring: Programming
'quick' found at position: 4
First character of 'Hello World': H
Last character of 'Hello World': d
Before clear: Temporary
After clear: '' (empty)
```

2.5 Random Number Functions (cstdlib header)

The `<cstdlib>` header provides functions for generating random numbers. These are useful for games, simulations, and testing.

Example: Random number generation

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

int main() {
    // Seed the random number generator with current time
    srand(time(0));

    // Generate random numbers
    cout << "5 random numbers (no range specified):" << endl;
    for (int i = 1; i <= 5; i++) {
        cout << rand() << " ";
    }
    cout << endl;

    // Random numbers in a specific range (1-10)
    cout << "\n10 random numbers between 1 and 10:" << endl;
    for (int i = 1; i <= 10; i++) {
        int random_num = (rand() % 10) + 1;
        cout << random_num << " ";
    }
    cout << endl;

    // Random numbers in range 50-100
    cout << "\n5 random numbers between 50 and 100:" << endl;
    for (int i = 1; i <= 5; i++) {
        int random_num = (rand() % 51) + 50;  // 51 different values
            (50-100 inclusive)
        cout << random_num << " ";
    }
    cout << endl;

    // Simulating dice rolls
    cout << "\nSimulating 10 dice rolls:" << endl;
    for (int i = 1; i <= 10; i++) {
        int dice = (rand() % 6) + 1;
        cout << "Roll " << i << ": " << dice << endl;
    }

    return 0;
}
```

Output:

```
5 random numbers (no range specified):
374651 985632 145289 652314 789456

10 random numbers between 1 and 10:
7 2 10 4 1 8 3 9 5 6

5 random numbers between 50 and 100:
73 91 55 68 84

Simulating 10 dice rolls:
Roll 1: 4
Roll 2: 2
Roll 3: 6
Roll 4: 1
Roll 5: 3
Roll 6: 5
Roll 7: 2
Roll 8: 6
Roll 9: 4
Roll 10: 1
```

2.6 Input/Output Functions

We've been using some built-in I/O functions throughout our examples. Here's a summary of the most common ones:

Example: Common I/O functions

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

int main() {
    // getline function for reading entire lines
    string full_name;
    cout << "Enter your full name: ";
    getline(cin, full_name);
    cout << "Hello, " << full_name << "!" << endl;

    // setw, setprecision, fixed (from iomanip)
    double price = 123.456789;
    cout << "\nFormatting examples:" << endl;
    cout << "Default: " << price << endl;
    cout << "Fixed with 2 decimals: " << fixed << setprecision(2) <<
       price << endl;
    cout << "Width of 10: |" << setw(10) << price << "|" << endl;

    // cin.ignore() to clear input buffer
    int number;
    cout << "\nEnter a number: ";
    cin >> number;
    cin.ignore();  // Clear the newline from buffer

    cout << "Enter another line of text: ";
    string text;
    getline(cin, text);
    cout << "You entered: " << text << endl;

    return 0;
}
```

Output:

```
Enter your full name: John Smith
Hello, John Smith!

Formatting examples:
Default: 123.457
Fixed with 2 decimals: 123.46
Width of 10: |    123.46|

Enter a number: 42
Enter another line of text: This is a test
You entered: This is a test
```

3 User-Defined Functions

3.1 What are User-Defined Functions?

User-defined functions are functions that you create to perform specific tasks in your program. While built-in functions handle common operations, user-defined functions handle the unique logic specific to your program. They allow you to organize your code into logical, reusable blocks.

A user-defined function consists of:

1. **Return Type:** The type of value the function returns (int, double, void, etc.)
2. **Function Name:** A descriptive name that indicates what the function does
3. **Parameter List:** Input values the function needs (can be empty)
4. **Function Body:** The code that performs the task
5. **Return Statement:** Returns a value to the caller (except for void functions)

3.2 Function Declaration and Definition

Before using a function, you need to declare it (tell the compiler it exists) and define it (provide the actual code). The declaration is also called a function prototype.

Example: Basic function structure

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

// Function declaration (prototype)
int add_numbers(int first_num, int second_num);

int main() {
    int result = add_numbers(5, 3);
    cout << "5 + 3 = " << result << endl;

    int x = 10, y = 20;
    int sum = add_numbers(x, y);
    cout << x << " + " << y << " = " << sum << endl;

    return 0;
}

// Function definition
int add_numbers(int first_num, int second_num) {
    int sum = first_num + second_num;
    return sum;
}
```

Output:

```
5 + 3 = 8
10 + 20 = 30
```

3.3 Types of Functions Based on Return Value

3.3.1 Functions that Return a Value

These functions compute something and return the result to the caller:

Example: Functions returning different types

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

// Function returning int
int calculate_square(int number) {
    return number * number;
}

// Function returning double
double calculate_average(double num1, double num2, double num3) {
    double sum = num1 + num2 + num3;
    double avg = sum / 3.0;
    return avg;
}

// Function returning bool
bool is_even(int number) {
    if (number % 2 == 0) {
        return true;
    } else {
        return false;
    }
}

// Function returning char
char get_grade(int score) {
    if (score >= 90) return 'A';
    else if (score >= 80) return 'B';
    else if (score >= 70) return 'C';
    else if (score >= 60) return 'D';
    else return 'F';
}
```

Example: Main function that calls the above user defined functions

```
int main() {
    // Using function that returns int
    int num = 7;
    int squared = calculate_square(num);
    cout << num << " squared = " << squared << endl;

    // Using function that returns double
    double avg = calculate_average(85.5, 92.0, 78.5);
    cout << "Average = " << avg << endl;

    // Using function that returns bool
    int test_num = 24;
    if (is_even(test_num)) {
        cout << test_num << " is even" << endl;
    } else {
        cout << test_num << " is odd" << endl;
    }

    // Using function that returns char
    int score = 85;
    char grade = get_grade(score);
    cout << "Score " << score << " gets grade " << grade << endl;
    return 0;
}
```

Output:

```
7 squared = 49
Average = 85.3333
24 is even
Score 85 gets grade B
```

3.3.2 Void Functions

Void functions perform an action but don't return a value. They're used for tasks like printing, modifying global variables, or performing operations that don't need to send back a result:

Example: Void functions

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

// Void function that prints a message
void print_welcome() {
    cout << "================================" << endl;
    cout << "   Welcome to the Calculator    " << endl;
    cout << "================================" << endl;
}

// Void function with parameters
void print_rectangle(int width, int height) {
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            cout << "* ";
        }
        cout << endl;
    }
}

// Void function that modifies output
void display_result(double num1, double num2, char operation) {
    cout << "Calculating: " << num1 << " " << operation << " " <<
       num2 << " = ";

    switch(operation) {
        case '+':
            cout << (num1 + num2) << endl;
            break;
        case '-':
            cout << (num1 - num2) << endl;
            break;
        case '*':
            cout << (num1 * num2) << endl;
            break;
        case '/':
            if (num2 != 0) {
                cout << (num1 / num2) << endl;
            } else {
                cout << "Error: Division by zero!" << endl;
            }
            break;
        default:
            cout << "Invalid operation!" << endl;
    }
}
```

Example: Main function that calls the above user defined functions

```
int main() {
    // Call void function with no parameters
    print_welcome();

    cout << "\nRectangle 5x3:" << endl;
    print_rectangle(5, 3);

    cout << "\nCalculations:" << endl;
    display_result(10, 5, '+');
    display_result(10, 5, '-');
    display_result(10, 5, '*');
    display_result(10, 5, '/');
    display_result(10, 0, '/');

    return 0;
}
```

Output:

```
================================
   Welcome to the Calculator
================================

Rectangle 5x3:
* * * * *
* * * * *
* * * * *

Calculations:
Calculating: 10 + 5 = 15
Calculating: 10 - 5 = 5
Calculating: 10 * 5 = 50
Calculating: 10 / 5 = 2
Calculating: 10 / 0 = Error: Division by zero!
```

3.4 Function Parameters and Arguments

Parameters are variables listed in the function declaration/definition. Arguments are the actual values passed to the function when it's called. C++ provides two main ways to pass arguments to functions:

4 Call by Value

4.1 Understanding Call by Value

Call by value is the default method of passing arguments to functions in C++. When you pass an argument by value, the function receives a **copy** of the actual value. This means:

- The function works with its own copy of the data

- Changes made to parameters inside the function do NOT affect the original variables
- The original values remain unchanged after the function call
- This is safer but uses more memory (for the copies)

Think of it like giving someone a photocopy of a document. They can write all over their copy, but your original document remains untouched.

4.2 How Call by Value Works

Example: Demonstrating call by value

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

// Function that tries to modify its parameter
void try_to_modify(int number) {
    cout << "Inside function - received value: " << number << endl;
    number = 100;  // Modifying the local copy
    cout << "Inside function - after modification: " << number <<
       endl;
}

// Function that swaps two numbers (doesn't work with call by value!)
void swap_values(int first, int second) {
    cout << "Inside swap - before: first = " << first << ", second =
       " << second << endl;
    int temp = first;
    first = second;
    second = temp;
    cout << "Inside swap - after: first = " << first << ", second = "
        << second << endl;
}

// Function that performs calculation
int calculate_factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++) {
        result = result * i;
    }
    n = 0;  // This doesn't affect the original variable
    return result;
}
```

Example: Main function that calls the above user defined functions

```
int main() {
    // Example 1: Trying to modify a value
    int my_number = 42;
    cout << "Before function call: my_number = " << my_number<< endl;
    try_to_modify(my_number);
    cout << "After function call: my_number = " << my_number << endl;
    cout << "Notice: Original value unchanged!\n" << endl;
    // Example 2: Attempting to swap
    int x = 10, y = 20;
    cout << "Before swap: x = " << x << ", y = " << y << endl;
    swap_values(x, y);
    cout << "After swap: x = " << x << ", y = " << y << endl;
    cout << "Notice: Original values NOT swapped!\n" << endl;
    // Example 3: Factorial calculation
    int num = 5;
    cout << "Number: " << num << endl;
    int fact = calculate_factorial(num);
    cout << "Factorial: " << fact << endl;
    cout << "Original number still: " << num << endl;
    return 0;
}
```

Output:

```
Before function call: my_number = 42
Inside function - received value: 42
Inside function - after modification: 100
After function call: my_number = 42
Notice: Original value unchanged!

Before swap: x = 10, y = 20
Inside swap - before: first = 10, second = 20
Inside swap - after: first = 20, second = 10
After swap: x = 10, y = 20
Notice: Original values NOT swapped!

Number: 5
Factorial: 120
Original number still: 5
```

4.3 When to Use Call by Value

Call by value is appropriate when:

- You want to protect the original data from being modified
- You're working with small data types (int, char, float, etc.)
- The function only needs to use the value, not modify it
- You want to ensure function doesn't have side effects

5 Understanding Call by Reference

5.1 What is Call by Reference?

Call by reference allows a function to work directly with the original variable rather than a copy. In C++, this is achieved using the reference operator (**&**).

Think of it like giving someone the actual key to your house instead of showing them a photograph. With the key, they can actually go inside and rearrange your furniture!

Key characteristics of call by reference:

- The function receives the memory address of the original variable
- Changes made inside the function **affect the original variable**
- No copy is made, saving memory
- More efficient for large data structures
- The function can modify the caller's variables
- Multiple values can be ”returned” by modifying reference parameters

5.2 Syntax of Call by Reference

To pass by reference, add an ampersand (**&**) after the parameter type in both the function declaration and definition:

Syntax:

```
return_type function_name(data_type &parameter_name) {
    // function body
}
```

The ampersand (&) is the reference operator. It tells C++ that this parameter should be a reference to the original variable, not a copy.

6 Basic Call by Reference Examples

6.1 Modifying a Single Value

Example: Successfully modifying a value with reference

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

void modify_value(int &number) {  // Note the & symbol
    cout << "Inside function, received: " << number << endl;
    number = 100;  // This WILL modify the original
    cout << "Inside function, after change: " << number << endl;
}

int main() {
    int original = 42;

    cout << "Before function call: " << original << endl;
    modify_value(original);
    cout << "After function call: " << original << endl;
    cout << "\nSuccess! The original value WAS changed!" << endl;

    return 0;
}
```

Output:

```
Before function call: 42
Inside function, received: 42
Inside function, after change: 100
After function call: 100

Success! The original value WAS changed!
```

6.2 The Swap Function That Actually Works

Example: Correct swap function using references

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

void swap_by_reference(int &first, int &second) {
    cout << "Inside swap, before: first = " << first
         << ", second = " << second << endl;

    int temp = first;
    first = second;
    second = temp;

    cout << "Inside swap, after: first = " << first
         << ", second = " << second << endl;
}

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

    cout << "Before swap: x = " << x << ", y = " << y << endl;
    swap_by_reference(x, y);
    cout << "After swap: x = " << x << ", y = " << y << endl;

    cout << "\nSuccess! The values WERE swapped!" << endl;

    return 0;
}
```

Output:

```
Before swap: x = 10, y = 20
Inside swap, before: first = 10, second = 20
Inside swap, after: first = 20, second = 10
After swap: x = 20, y = 10

Success! The values WERE swapped!
```

6.3 Understanding Memory: Call by Value vs Call by Reference

Example: Visualizing memory addresses

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

void by_value(int num) {
    cout << "Inside by_value function:" << endl;
    cout << "  Address of parameter: " << &num << endl;
    cout << "  Value: " << num << endl;
    num = 999;
}

void by_reference(int &num) {
    cout << "Inside by_reference function:" << endl;
    cout << "  Address of parameter: " << &num << endl;
    cout << "  Value: " << num << endl;
    num = 999;
}

int main() {
    int x = 42;

    cout << "Original variable in main:" << endl;
    cout << "  Address of x: " << &x << endl;
    cout << "  Value: " << x << endl << endl;

    cout << "--- Call by Value ---" << endl;
    by_value(x);
    cout << "After by_value, x = " << x << endl;
    cout << "Notice: Different addresses = different variables\n" <<
       endl;

    cout << "--- Call by Reference ---" << endl;
    by_reference(x);
    cout << "After by_reference, x = " << x << endl;
    cout << "Notice: Same addresses = same variable!" << endl;

    return 0;
}
```

Output:

```
Original variable in main:
  Address of x: 0x7ffd5e8a3c4c
  Value: 42

--- Call by Value ---
Inside by_value function:
  Address of parameter: 0x7ffd5e8a3c2c
  Value: 42
After by_value, x = 42
Notice: Different addresses = different variables

--- Call by Reference ---
Inside by_reference function:
  Address of parameter: 0x7ffd5e8a3c4c
  Value: 42
After by_reference, x = 999
Notice: Same addresses = same variable!
```

7 Practical Applications of Call by Reference

7.1 Returning Multiple Values

One of the most powerful uses of call by reference is returning multiple values from a function. Normally, a function can only return one value. With references, we can ”return” multiple values by modifying reference parameters.

Example: Calculating multiple statistics

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

void calculate_stats(int val1, int val2, int val3,
                     int &sum, double &average, int &max) {
    // Calculate sum
    sum = val1 + val2 + val3;

    // Calculate average
    average = sum / 3.0;

    // Find maximum
    max = val1;
    if (val2 > max) max = val2;
    if (val3 > max) max = val3;
}

int main() {
    int a = 85, b = 92, c = 78;
    int total, maximum;
    double avg;

    cout << "Input values: " << a << ", " << b << ", " << c << endl;

    // Pass variables by reference to get results
    calculate_stats(a, b, c, total, avg, maximum);

    cout << "\nResults:" << endl;
    cout << "Sum: " << total << endl;
    cout << "Average: " << avg << endl;
    cout << "Maximum: " << maximum << endl;

    return 0;
}
```

Output:

```
Input values: 85, 92, 78

Results:
Sum: 255
Average: 85
Maximum: 92
```

7.2 Time Conversion

Example: Converting seconds to hours, minutes, seconds

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

void convert_seconds(int total_seconds, int &hours, int &minutes, int
    &seconds) {
    hours = total_seconds / 3600;
    minutes = (total_seconds % 3600) / 60;
    seconds = total_seconds % 60;
}

int main() {
    int total_sec = 7265;  // 2 hours, 1 minute, 5 seconds
    int hrs, mins, secs;

    cout << "Converting " << total_sec << " seconds..." << endl;

    convert_seconds(total_sec, hrs, mins, secs);

    cout << "Result: " << hrs << " hours, "
         << mins << " minutes, " << secs << " seconds" << endl;

    // Try another conversion
    total_sec = 3661;
    convert_seconds(total_sec, hrs, mins, secs);
    cout << "\n" << total_sec << " seconds = " << hrs << " hours, "
         << mins << " minutes, " << secs << " seconds" << endl;

    return 0;
}
```

Output:

```
Converting 7265 seconds...
Result: 2 hours, 1 minutes, 5 seconds

3661 seconds = 1 hours, 1 minutes, 1 seconds
```

7.3 Division with Quotient and Remainder

Example: Performing division with both results

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

void divide_with_remainder(int dividend, int divisor,
                           int &quotient, int &remainder) {
    if (divisor == 0) {
        cout << "Error: Division by zero!" << endl;
        quotient = 0;
        remainder = 0;
        return;
    }

    quotient = dividend / divisor;
    remainder = dividend % divisor;
}

int main() {
    int dividend = 17, divisor = 5;
    int quot, rem;

    divide_with_remainder(dividend, divisor, quot, rem);

    cout << dividend << " divided by " << divisor << ":" << endl;
    cout << "Quotient: " << quot << endl;
    cout << "Remainder: " << rem << endl;
    cout << "Verification: " << divisor << " x " << quot << " + "
         << rem << " = " << (divisor * quot + rem) << endl;

    return 0;
}
```

Output:

```
17 divided by 5:
Quotient: 3
Remainder: 2
Verification: 5 x 3 + 2 = 17
```

8 Comparing Call by Value and Call by Reference

8.1 Side-by-Side Comparison

Example: Direct comparison of both methods

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

// Call by value version
void square_by_value(int number) {
    cout << "  Inside by_value: ";
    number = number * number;
    cout << number << endl;
}

// Call by reference version
void square_by_reference(int &number) {
    cout << "  Inside by_reference: ";
    number = number * number;
    cout << number << endl;
}

int main() {
    cout << "=== Comparing Call by Value and Call by Reference ==="
       << endl;

    // Test with call by value
    int num1 = 5;
    cout << "\nCall by Value Test:" << endl;
    cout << "Before: num1 = " << num1 << endl;
    square_by_value(num1);
    cout << "After: num1 = " << num1 << " (unchanged)" << endl;

    // Test with call by reference
    int num2 = 5;
    cout << "\nCall by Reference Test:" << endl;
    cout << "Before: num2 = " << num2 << endl;
    square_by_reference(num2);
    cout << "After: num2 = " << num2 << " (changed!)" << endl;

    return 0;
}
```

Output:

```
=== Comparing Call by Value and Call by Reference ===

Call by Value Test:
Before: num1 = 5
  Inside by_value: 25
After: num1 = 5 (unchanged)

Call by Reference Test:
Before: num2 = 5
  Inside by_reference: 25
After: num2 = 25 (changed!)
```

8.2 Mixed Parameters

You can mix call by value and call by reference in the same function. This gives you fine control over which parameters can be modified.

Example: Mixing value and reference parameters

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

// First two parameters by value, last two by reference
void process_data(int input1, int input2, int &output1, int &output2)
    {
    cout << "Inside function:" << endl;
    cout << "  Received input1 = " << input1 << ", input2 = " <<
       input2 << endl;

    // These modifications only affect local copies
    input1 = input1 * 2;
    input2 = input2 * 2;

    // These modifications affect the original variables
    output1 = input1 + 100;
    output2 = input2 + 100;

    cout << "  Modified input1 = " << input1 << ", input2 = " <<
       input2 << endl;
    cout << "  Set output1 = " << output1 << ", output2 = " <<
       output2 << endl;
}

int main() {
    int a = 10, b = 20, c = 0, d = 0;

    cout << "Before function:" << endl;
    cout << "  a = " << a << ", b = " << b << ", c = " << c << ", d =
        " << d << endl;

    process_data(a, b, c, d);

    cout << "\nAfter function:" << endl;
    cout << "  a = " << a << " (unchanged - passed by value)" << endl
       ;
    cout << "  b = " << b << " (unchanged - passed by value)" << endl
       ;
    cout << "  c = " << c << " (changed - passed by reference)" <<
       endl;
    cout << "  d = " << d << " (changed - passed by reference)" <<
       endl;

    return 0;
}
```

Output:

```
Before function:
  a = 10, b = 20, c = 0, d = 0
Inside function:
  Received input1 = 10, input2 = 20
  Modified input1 = 20, input2 = 40
  Set output1 = 120, output2 = 140

After function:
  a = 10 (unchanged - passed by value)
  b = 20 (unchanged - passed by value)
  c = 120 (changed - passed by reference)
  d = 140 (changed - passed by reference)
```

Key Takeaways:

- **Call by Value:** Function works with a copy; original unchanged
- **Call by Reference:** Function works with original; changes affect caller
- **Const Reference:** Efficient like reference, safe like value
- Use `&` to create reference parameters
- Reference parameters allow returning multiple values
- Choose the appropriate method based on your needs
- Const references are best for large read-only data
- Never return references to local variables