SWEG2102 Fall 2026

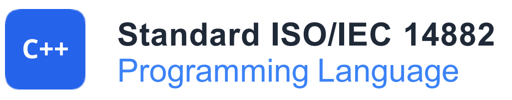
Fundamentals of Programming II

Lecture 03

Modular Programming (Part II)
Advanced Functions

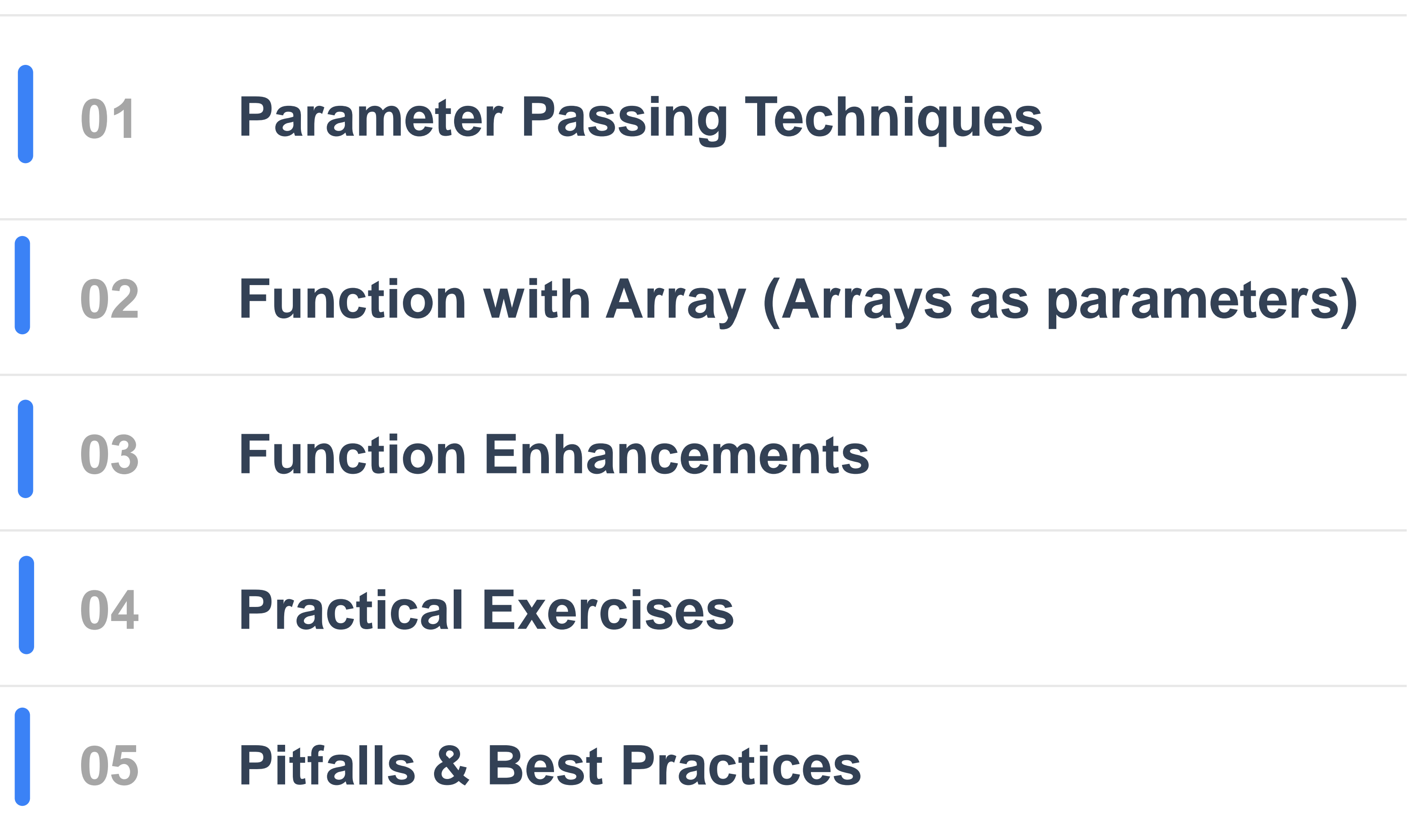
Chere Lemma (M.Tech)
Lecturer, Dept. of Software Engineering

Addis Ababa Science and Technology University (AASTU)
Addis Ababa, Ethiopia



Outline

Topics Covered



Learning Objectives

By the end of this lecture, you will be able to:

- Explain and differentiate parameter passing techniques.
- Use arrays as function parameters correctly.
- Apply function enhancements for cleaner and more efficient code.
- Identify and avoid common function pitfalls following best practices.
- Build small programs using both basic and advanced function concepts.

Parameter Passing Techniques

1. Parameter Passing

Core Concept

Parameter passing is the mechanism used in programming to transfer **data (arguments)** from the calling function to the called function.

Parameters

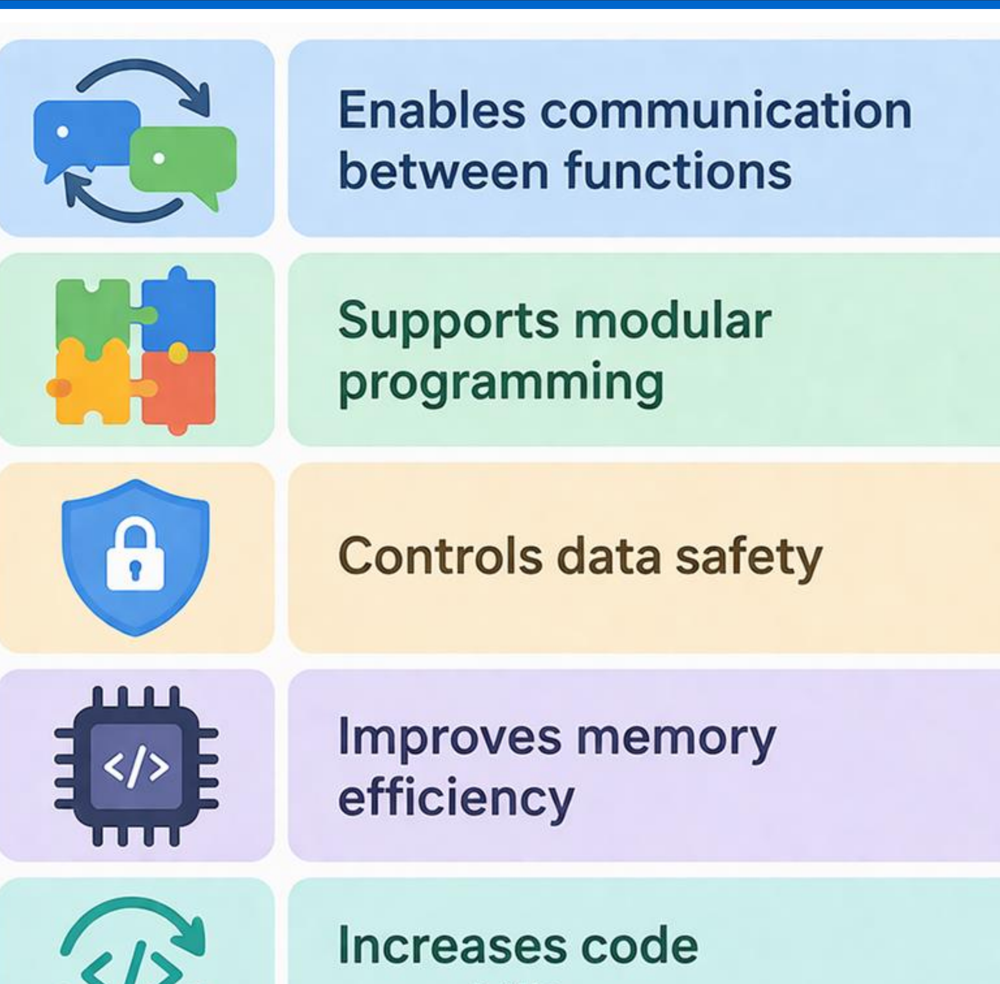
- Refers to the variables listed in the function definition/declaration
- They define what kind of data the function expects (placeholders).
- Also called **Formal Parameters**

Arguments

- Refers to the actual values passed to the function when it is called
- Provide actual data, not a placeholder
- Also referred as **Actual Parameters**

Cont’d

Why Parameter Passing is Important?



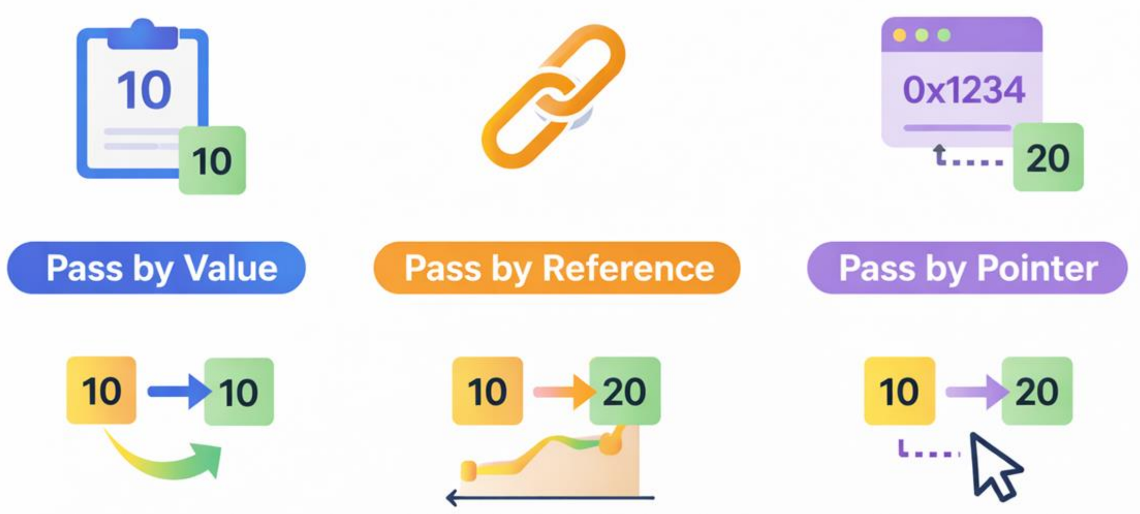
Source: *AI-generated image (ChatGPT 2026)*

Cont'd

Parameter Passing Techniques

- There are **three common techniques** of parameter passing.
- The choice of parameter passing technique determines **how memory is managed** and whether the o**riginal data can be modified**.
- In other terms, it also affects **program performance and data safety**.

Three main techniques



Source: *AI-generated image (ChatGPT 2026)*

1.1. Pass by Value

Core Concept

- A copy of the actual argument is passed to the function and created in the formal parameter's memory location
  - ✓ The function works entirely on this **copy**
  - ✓ Original variable remains unchanged
- 👉 Safe but memory-consuming for large data

Syntax & Example (C++)

```
void update (int x) {
      x = x + 10;
}
int main() {
            int a = 5;
            update (a);
            cout << a;          // Output: 5
 }
```

When to Use and not

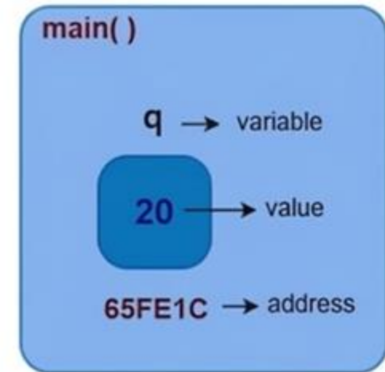
- Small Primitives data types
- Read-Only Data - When original data must be protected
- Avoid for large objects (inefficient copying)

Cont’d

Memory Visualization

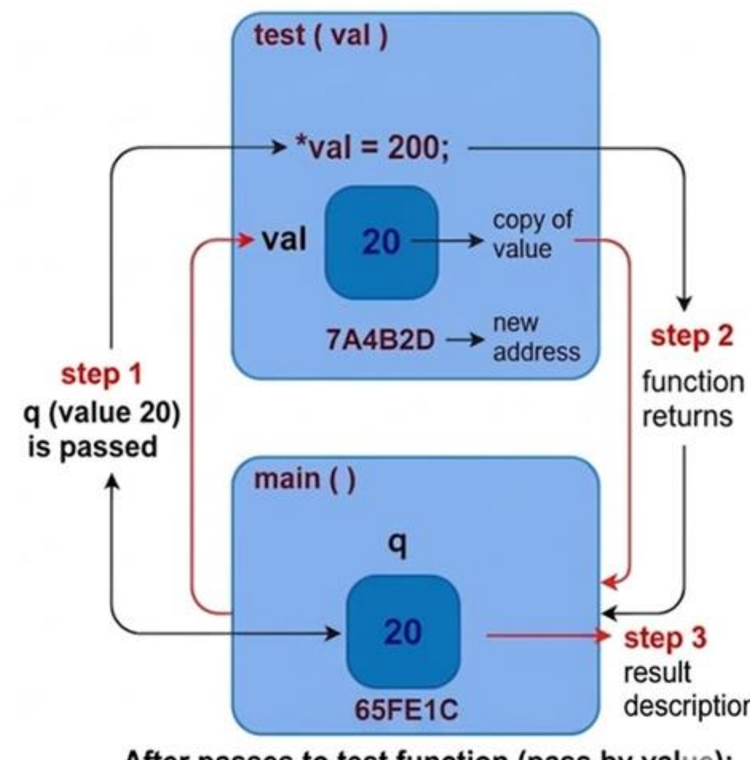
```cpp
#include <iostream>
using namespace std;

// Pass by Value function
void test(int val) {
    cout<<"Inside function: ";
    cout<< val<<endl;

    val = 200; //changes only local copy

    cout<<"After change: ";
    cout<< val<<endl;
}

int main() {
    int q = 20;

    cout<<"Before function call: ";
    cout<< q << endl;

    test(q); // original value is passed

    cout<<"After function call: ";
    cout<<q<<"(unchanged)"<< endl;

    return 0;
}
```

Before calling the function



Initial value of q = 20

After calling the function



After passes to test function (pass by value):
Inside test function: **val = 200** (modification complete)
Back in main: **q = 20** (q is unchanged)

Source: *AI-generated image (Gemini 3)*

1.2. Pass by Reference

Core Concept

- The function receives an **alias (reference or another name)** for the original variable.
  - ✓ No copy is created
  - ✓ Both variables refer to **the same memory** location and no new memory is allocated
  - ✓ Changes **directly affect** original variable

When to Use and not

- When function must modify original variable
- To pass large objects efficiently without copying
- When returning multiple outputs indirectly

Syntax & Example (C++)

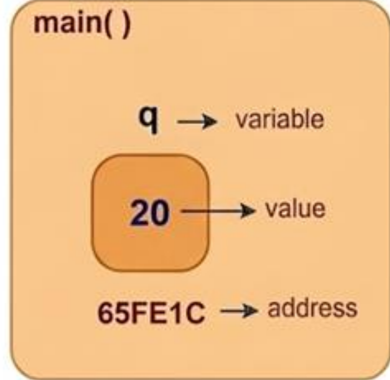
```
// '&' denotes a reference
void increment (int &n) {
      n = ++n *2;
}
int main() {
            int x = 10;
            increment (x);
            cout << x;  // x becomes 22
 }
```

Cont'd

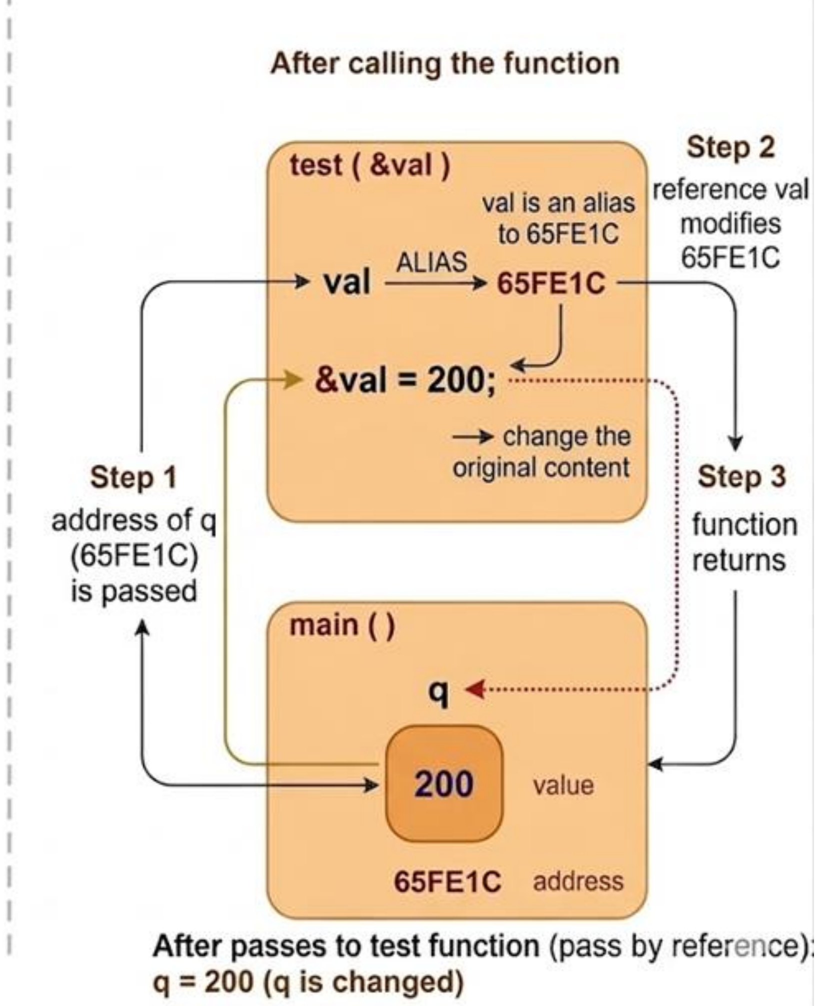
Memory Visualization

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

// Pass by Reference function
void test(int &val) {
    cout << "Inside function: "<< val << endl;

    //Directly modifies the original variable
    val = 200;

    cout<<"After change: "<< val << endl;
}

int main() {
    int q = 20;

    cout<<"Before function call: ";
    cout<< q << endl;

    test(q); // The address of q is passed

    cout<<"After function call: "<< q;
    cout<< " (changed)" << endl;

    return 0;
}
```

Before calling the function



Initial value of q = 20



Source: *AI-generated image (Gemini 3)*

1.3. Pass by Pointer

Core Concept

- The **memory address** of the variable is passed to the function.
  - ✓ The function **uses a pointer** to "point" to the original variable's location and access it.
  - ✓ Like **pass by reference**, it allows the function to modify the original data,
  - ✓ It requires **explicit dereferencing**.

When to Use and not

- When working with dynamic memory (new, delete)
- When dealing with arrays and buffers
- Low-level memory manipulation

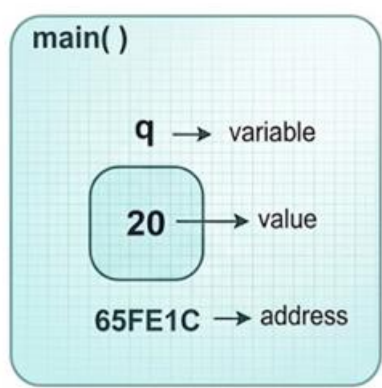
Syntax & Example (C++)

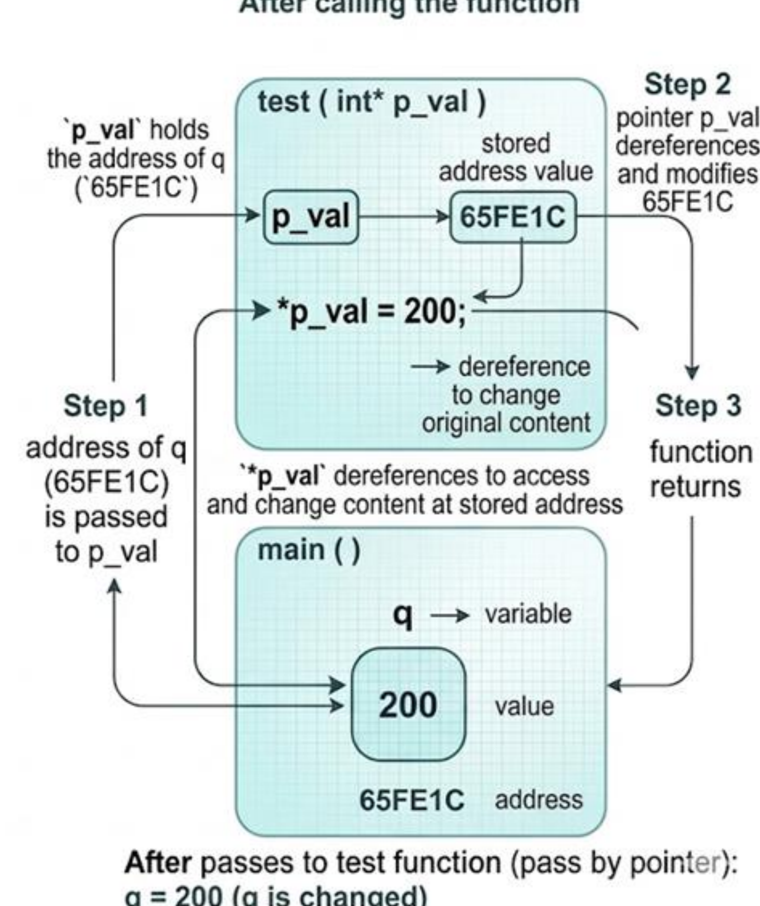
```cpp
// '&' denotes a reference and
// '*' refers to pointer / dereferencing
void update (int *x) {
    *x = *x + 10;
}
int main() {
        int a = 5;
        update (&a);
        cout << a;  // output: 15
}
```

Cont'd

Memory Visualization

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

// Pass by pointer function
void test(int *val) {
    cout<<"Inside function: "<< *val<< endl;

    //Directly modifies the original variable
    *val = 200;

    cout<<"After change: "<< *val << endl;
}

int main() {
    int q = 20;

    cout<<"Before function call: ";
    cout<< q << endl;

    test(&q); // The address of q is passed

    cout<<"After function call: ";
    cout<< q <<" (changed)" << endl;

    return 0;
}
```





Source: *AI-generated image (Gemini 3)*

Cont'd

Parameter Passing Comparison

Pass by Value

10 → Function (param) → 10

Original — Copy

A **copy** of the actual value is passed to the function.

- Changes inside function do not affect original
- Safer (data is protected)
- Uses more memory (for the copy)

Syntax: `void f(int x);`

Pass by Reference

10 → Function (param) → 20

Original — Modified

A **reference** (alias) of the original variable is passed.

- Changes inside function affect the original
- Efficient (no copying of data)
- Direct access to actual variable

Syntax: `void f(int &x);`

Pass by Pointer

10 → Function (*ptr) → 20

Original — Modified

A memory address (**pointer**) of the variable is passed.

- Passes the address of the variable
- Function can modify original data using *ptr
- Used in low-level & system programming

Syntax: `void f(int *x);`

Source: *AI-generated image (Gemini 3)*

Quick Check: Pause & Predict

1. Given **void change(int &x) { x = 50},** What will be the value of **y** after function call **change(y),** where initially **y =13?**

***Answer: 50** (Since x is a reference parameter, it directly modifies the original variable y)*

2. Given **void change(int *a) { *a = + 2 },** What will be printed with function call **change(a),** where **a =3?**

***Answer: compiler error** (The function expects a pointer (int*), but an integer value (int) is passed)*

3. Given the below code segment, What will be the value of **a & b** with function call **swap(a, b),** where **a = 3, b = 5?**

```
void swap (int &x, int y) {
    int temp = *x;
    *x = y;
    y = temp
}
```

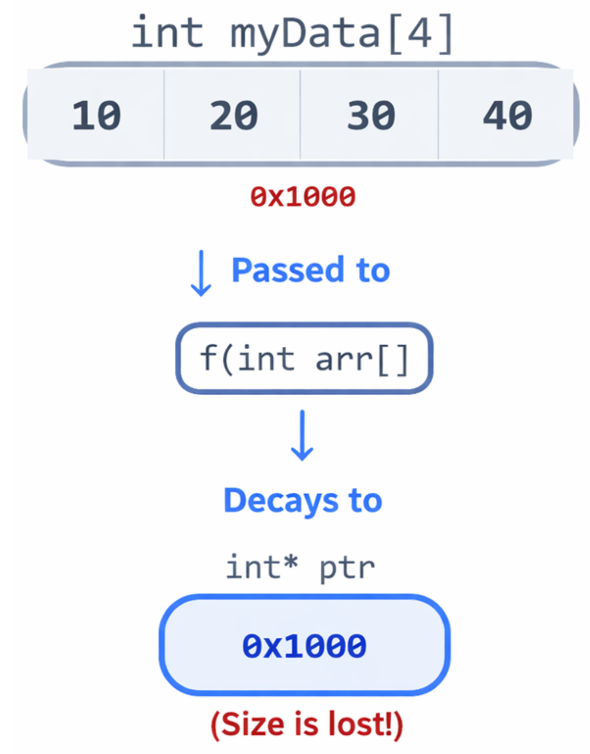
Answer: a = 5, b = 3
(Because both parameters are passed by reference)

2. Arrays as a Parameter

What does it mean?

- **Passing an array** to a function means sending the array's data so the function can process multiple values.
  - 👉 Instead of **passing one value**, we pass a collection of values.
- When **array** passed to a function, an **array T arr[N]** implicitly decays into a **pointer T*** to its first element.
- The **size information** is completely lost in the parameter and the **function** only knows where the **array starts**, **not where it ends**.

Decay Visualization



Cont'd

Syntax and Example

- void function_name (type arrayName[], int arraySize);

Or

- void function_name (type* arrayName, int arraySize);

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

void printArray(int arr[], int size) {
    for(int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
}

int main() {
    int numbers[] = {1, 2, 3, 4, 5};

    printArray(numbers, 5);

    return 0;
}
```

Cont'd

Passing Array by Reference

When an array is **passed by reference**, the function receives the **entire array** as it is, instead of a pointer to its first element.

- ✅ This **prevents array decay** (i.e., it does NOT turn into a pointer)
- ✅ The **size of the array is preserved**
- ✅ The compiler knows the array size at **compile time**

Syntax:

```
void funcName (type (&array)[size]);
```

Example:

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

void print(int (&arr)[5]) {
    for(int i = 0; i < 5; i++) {
        cout << arr[i] << " ";
    }
}

int main() {
    int data[5] = {1, 2, 3, 4, 5};

    print(data); // size is known (5)
}
```

Function Enhancements

3.1. Default Arguments

Concept

Allows functions to use **default values** if arguments are omitted by the caller.

👉 *It makes functions more flexible and reduces the need for multiple versions.*

Syntax

👉 return_type function_name (type param = default_value);

Rules

- Defaults **must be the trailing** (rightmost) parameters in the function signature
- Specify the default value only once, usually within the **function declaration/prototype**
- Values are resolved statically at **compile time**

Cont'd

Example: Default Argument Scenarios

Case 1: No Argument is Passed

```
void temp(int = 10, float = 8.8);

int main() {
  temp();
}

void temp(int i, float f) {
  cout << i << "," << f << ";" << endl;
}
```

DEFAULT (10)
DEFAULT (8.8)
RESULT: 10, 8.8;

Case 2: First Argument is Passed

```
void temp(int = 10, float = 8.8);

int main() {
  temp(6);
}

void temp(int i, float f) {
  cout << i << "," << f << ";" << endl;
}
```

PASSED (6)
DEFAULT (8.8)
RESULT: 6, 8.8;

Case 3: All Arguments are Passed

```
void temp(int = 10, float = 8.8);

int main() {
  ...
  temp(6, -2.3);
  ...
}

void temp(int i, float f) {
  cout << i << "," << f << ";" << endl;
}
```

PASSED (6)
PASSED (-2.3)
RESULT: 6, -2.3;

Case 4: Second Argument is Passed (Attempt)

```
void temp(int = 10, float = 8.8);

int main() {
  ...
  temp(3.4);
  ...
}

void temp(int i, float f) {
  cout << i << "," << f << ";" << endl;
}
```

Warning: Argument coerced from float to int
COERCED (3.4 → 3)
Error
COMPILER ERROR: Argument mismatch

Source:
https://www.programiz.com/cpp-programming/default-argument

Recreate with AI (Gemini 3)

3.2. Function Overloading

Concept

- Refers to functions with the **same name** but **different parameter lists** (signatures).
  - 👉 Compiler decides which function to call based on **Number and Type** of parameters
- Improves expressiveness—use one meaningful name for similar conceptual actions.

Syntax

- 👉 return_type function_name (type1);
- 👉 return_type function_name (type1, type2);

Overloading Rules

- Functions must differ in **Number** OR type of parameters
- ❌ Cannot overload only by return type

Cont'd

Code Examples

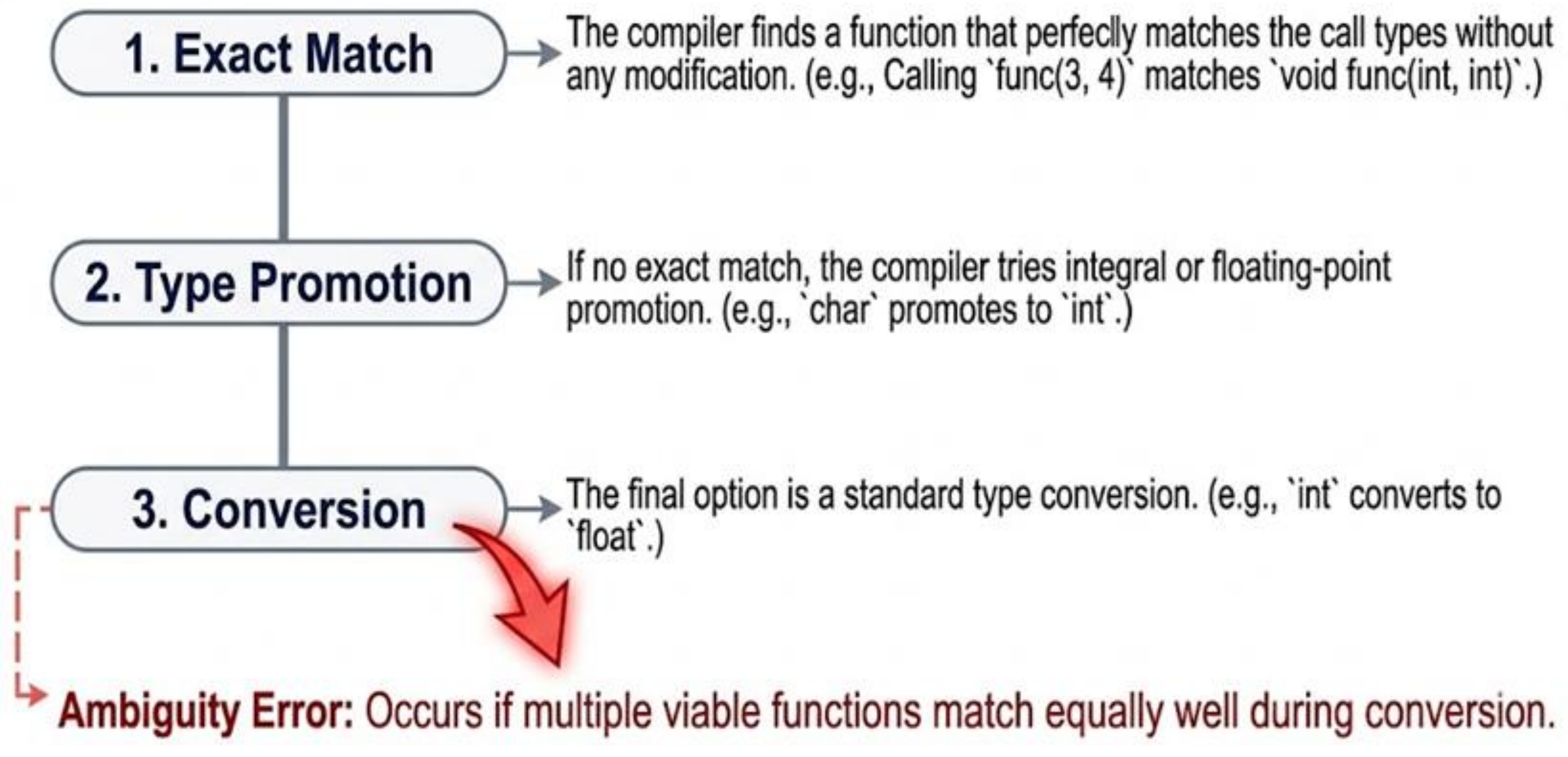
```cpp
#include <iostream>
using namespace std;

// function with 2 parameters
void display(int var1, double var2) {
    cout << "Integer number: " << var1;
    cout << " and double number: " << var2 << endl;
}

// function with double type, single parameter
void display(double var) {
    cout << "Double number: " << var << endl;
}

// function with int type, single parameter
void display(int var) {
    cout << "Integer number: " << var << endl;
}
```

```cpp
int main() {

    int a = 5;
    double b = 5.5;

    // call function with int type parameter
    display(a);

    // call function with double type parameter
    display(b);

    // call function with 2 parameters
    display(a, b);

    return 0;
}
```

Source: *https://www.programiz.com/cpp-programming/function-overloading*

Cont'd

Compiler Resolution Flow



3.3. Inline Functions

Concept

- An **inline function** tells the compiler to **replace the function call** with the actual function code.

👉 Eliminates function call overhead → faster execution.

When to Use & Caveats

- Ideal for very small (tiny), frequently executed functions to avoid call overhead
  - ✓ e.g., getters, math operations.
- Performance-critical code
  - ✓ Inlining large functions copies code everywhere, increasing executable size.

❌ Avoid using with:

- *Large functions*
- *Recursive functions*
- *Functions with loops/complex logic*

Cont'd

Code Examples

```
inline void displayNum(int num) {
  cout << num << endl;
}

int main() {

  displayNum(5);

  displayNum(8);

  displayNum(666);
}
```

Compilation

```
inline void displayNum(int num) {
  cout << num << endl;
}

int main() {

  cout << 5 << endl;

  cout << 8 << endl;

  cout << 666 << endl;
}
```

Source: *https://www.programiz.com/cpp-programming/inline-function*

Summary of Function Enhancement

DEFAULT ARGUMENTS

SYNTAX:

```
void f(int x = 0);
```

BEHAVIOR:

- ✓ Supplies missing args
- ✓ Trailing params only
- ✓ Resolved at compile time

USE WHEN:

- ➢ Adding optional config
- ➢ When parameters are optional
- ➢ To reduce multiple similar functions
- ➢ For cleaner and shorter function calls

FUNCTION OVERLOADING

SYNTAX:

```
void f(int);
void f(double);
```

BEHAVIOR:

- ✓ Different signatures
- ✓ Same function name
- ✓ Resolved by param match

USE WHEN:

- ➢ Same conceptual action/operation, but different data types/arity
- ➢ Improves readability and reuse
- ➢ Reduces need for different function names

INLINE FUNCTIONS

SYNTAX:

```
inline int getX();
```

BEHAVIOR:

- ✓ Substitutes call with body
- ✓ Reduces call overhead
- ✓ Compiler optimization hint

USE WHEN:

- ➢ Very small / hot functions (e.g., math operations)
- ➢ Frequently called functions
- ➢ Performance-critical code

Source: *AI-generated image (Gemini 3)*

Quick Check: Pause & Predict

1. **What's wrong here?**

```
void g(int a=1, int b);
```

Answer: Defaults must be trailing.

- *Parameter **b** lacks a default value after the defaulted parameter **a**.*

2. **Are these valid overloads?**

```
int height(int);
double height(double = 3.14);
```

Answer: Yes, they have different signatures

- *However, passing types requiring conversions (like long) can cause ambiguity at the call site*

3. **Which function uses pass by reference?**

A. `void func (int x);`

B. `void func (int *x);`

C. `void func (int &x);`

D. `void func (const int x);`

Answer: C

- ***&** indicates reference parameter*

Practical Exercises

4. Practice Exercise

Exercise #1 — Student Score Analyzer

Objective

Build a modular student sc**ore analyzer system** that demonstrates:

- Different **parameter passing techniques**
- Use of **default arguments**
- Implementation of **function overloading**

Requirements

1. Create overload function named **calcAvergae ()**
   - *Return type: double*
   - *Parameters: int type*
   - *Overloading**:** number of **parameter** (2, 3, 4)*
2. Create function with default arguments
   - *void displayResult(double avg, char grade = “F");*
3. Create functions that compute bonus:
   - *Pass by value: void addBonus (int score);*
   - *Pass by value: void addBonusRef (int score);*

Cont’d

Solution : Student Score Analyzer

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

// ====== Function Overloading ===========
double calculateAverage(int a, int b) {
    return (a + b) / 2.0;
}

double calculateAverage(int a, int b, int c) {
    return (a + b + c) / 3.0;
}

double calculateAverage(int a, int b, int c, int d) {
    return (a + b + c + d) / 4.0;
}

// ================ Default Argument =================
void displayResult(double avg, char grade = "F") {
    cout << "Average Score: " << avg << endl;
    cout << "Grade: " << grade << endl;
}
```

```cpp
// ============== Parameter Passing ==============
// Pass by Value
void addBonus(int score) {
    score += 5;
    cout<<"[Pass by Value] Score inside function: "
    cout<< score << endl;
}

// Pass by Reference
void addBonusRef(int &score) {
    score += 5;
    cout<<"[Pass by Reference] Score inside function:
    cout<< score << endl;
}

// Pass by Pointer
void addBonusPtr(int *score) {
    *score += 5;
    cout<<"[Pass by Pointer] Score inside function: "
    cout<< *score << endl;
}
```

Cont'd

Solution : Student Score Analyzer

```cpp
int main() {
    int a, b, c, d;
    int choice;
    cout << "Enter number of scores (2-4): ";
    cin >> choice;
    double avg;

    if (choice == 2) {
        cout << "Enter 2 scores: ";
        cin >> a >> b;
        avg = calculateAverage(a, b);
    } else if (choice == 3) {
        cout << "Enter 3 scores: ";
        cin >> a >> b >> c;
        avg = calculateAverage(a, b, c);
    } else if (choice == 4) {
        cout << "Enter 4 scores: ";
        cin >> a >> b >> c >> d;
        avg = calculateAverage(a, b, c, d);
    }
    else {
        cout << "Invalid choice!" << endl;
        return 0;
    }
```

```cpp
    // Display with default argument
    displayResult(avg);

    // ======== Demonstrate Parameter Passing ============
    int score = 50;
    cout << "\nOriginal Score: " << score << endl;

    addBonus(score);
    cout << "After Pass by Value: " << score << endl;

    addBonusRef(score);
    cout << "After Pass by Reference: " << score << endl;

    addBonusPtr(&score);
    cout << "After Pass by Pointer: " << score << endl;

    return 0;
}
```

5. Common Pitfalls and Best Practices

Common Pitfalls

- ✕ Passing wrong argument type
- ✕ Mixing default arguments with function overloading
- ✕ Wrong order in default argument definition
- ✕ Overloading function only by return type
- ✕ Redefining default values in multiple places
- ✕ Using **inline** for large functions and with recursion or complex logic
- ✕ Forgetting **array size** while using array as argument in function

Cont'd

Best Practices

- ✓ Use pass **by value** for small data and safety
- ✓ Use pass **by reference** for large data or modification required
- ✓ Always validate pointers before dereferencing
- ✓ Define defaults arguments from **right to left**
- ✓ Declare **default values** in function prototype
- ✓ Avoid using inline with recursion function, and function that has large logic
- ✓ Avoid excessive overloading

References

Textbooks

- **C++ How to Program** [10th edition], Deitel, P. & Deitel, H., Global Edition, Global Edition (2017).
- **Problem Solving With C++** [10th edition], Walter Savitch, University of California, San Diego, 2018.

Reference Books

- **Programming: Principles and Practice Using C++** by Bjarne Stroustrup, Addison-Wesley, 2014.
- **An Introduction to Programming with C++** (8th Edition), Diane Zak, Cengage Learning, 2016

Online Resources

- https://www.geeksforgeeks.org/cpp/c-plus-plus/
- https://www.w3schools.com/cpp/default.asp
- https://programiz.pro/resources/cpp
- https://www.hackerrank.com/domains/cpp
- https://cplusplus.com/doc/tutorial/

Study Tip: *Don't just read the code! Retype the examples from these slides and resources into your IDE, compile them, and modify them to see what happens.*

Thank You!

Chere Lemma (M.Tech)
Lecturer, Dept. of Software Engineering

Addis Ababa Science and Technology University (AASTU)
Addis Ababa, Ethiopia