Function Overloading

Narrative/Analogy: "Think of a restaurant menu. You might see 'Burger' listed multiple times. One is a 'Cheeseburger', another is a 'Veggie Burger'. Same name ('Burger'), different details (ingredients). The chef knows which one to make based on what you order. Function overloading works similarly."

Definition: "Function overloading allows us to create multiple functions with the same name, but different parameters (type, number, or order). The compiler picks the right one based on how you call it."

```
 #include <iostream>
// Function 1: Takes an int
void print(int x) {
   std::cout << "Int: " << x << std::endl;
}
// Function 2: Takes a double (different type)
void print(double x) {
   std::cout << "Double: " << x << std::endl;
}
// Function 3: Takes two ints (different number)
void print(int x, int y) {
   std::cout << "Two Ints: " << x << ", " << y << std::endl;
}

int main() {
   print(5);      // Calls print(int) - prints "Int: 5"
   print(3.14);   // Calls print(double) - prints "Double: 3.14"
   print(1, 2);   // Calls print(int, int) - prints "Two Ints: 1, 2"
   return 0;
}
```

Key Point: "The functions must differ in their parameter list. Return type alone isn't enough!"

Professional Note: Explain how the compiler resolves which function to call (function signature).

Engagement: "What if I called `print(5.0f)`? Which function would be called?" (Answer: Depends on if there's a `float` version, otherwise might be promoted to `double`). "What if I had `void print(int x)` and `void print(const int x)`?" (Answer: Compilation error - only top-level `const` doesn't differentiate signatures).

Potential Humor: "Function overloading is like having a friend named Alex. Depending on the context (Alex the chef, Alex the mechanic), you know who you're talking to. Just don't get them confused at a party!"

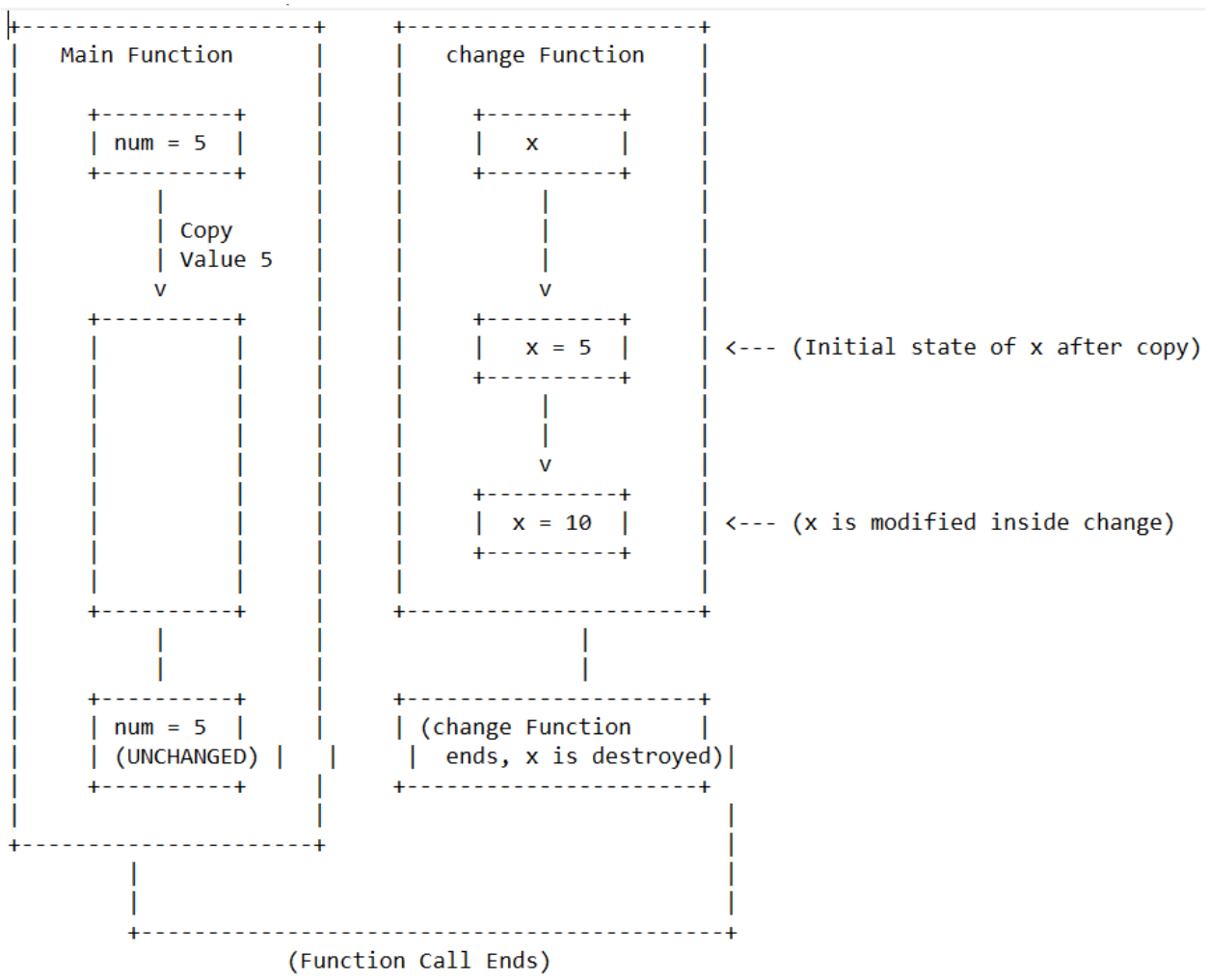
Call by Value

Narrative/Analogy: "Imagine you want someone to edit an important document for you. What's the safest way? Give them a photocopy. They can scribble all over the copy, but your original stays pristine. This is Call by Value."

Definition: "When you pass a variable by value, the function receives a copy of the variable's data. Any changes made inside the function only affect the copy."

Visual Diagram 1: Call by Value

Draw two columns: "Main Function" and "change Function".
In "Main": Draw a box `num = 5`.
In "change": Draw a box `x` (initially empty).
Show an arrow from `num` to `x` with a label "Copy Value 5".
Inside `change`: Show `x = 10`.
Back in "Main": Show `num` is still `5`.



```
#include <iostream>
void change(int x) { // Receives a COPY of the value
    std::cout << "Inside change(), x was: " << x << std::endl; // Prints 5
    x = 10; // Modifies the COPY
    std::cout << "Inside change(), x is now: " << x << std::endl; // Prints 10
}

int main() {
    int num = 5;
    std::cout << "Before change(), num is: " << num << std::endl; // Prints 5
    change(num); // Pass the VALUE of num
    std::cout << "After change(), num is: " << num << std::endl; // Still prints 5!
    return 0;
}
```

Key Point: "The original variable in the caller (`num`) is unchanged."

Professional Note: Mention overhead of copying (especially for large objects).

Engagement: "Quick Concept Check (Verbal): In the `Easy 1` problem, what gets printed? Why?" (Answer: 10, because the function worked on a copy).

Potential Humor: "Call by Value is safe, but it's like sending your sibling to the store with a list, but they memorize it instead of taking the paper. If they forget something, your original list at home is useless."

Call by Reference

Narrative/Analogy: "Okay, the photocopy method was safe, but what if you want the editor to change the original document? You'd give them the original file and tell them, 'Here, work on this directly.' That's Call by Reference. Or, think of giving someone your bank account number – any transaction they make affects your account directly."

Definition: "When you pass a variable by reference (using `&` in the parameter), the function gets direct access to the original variable's memory location. Changes made inside the function affect the original variable."

Visual Diagram 2: Call by Reference

Draw two columns: "Main Function" and "change Function".

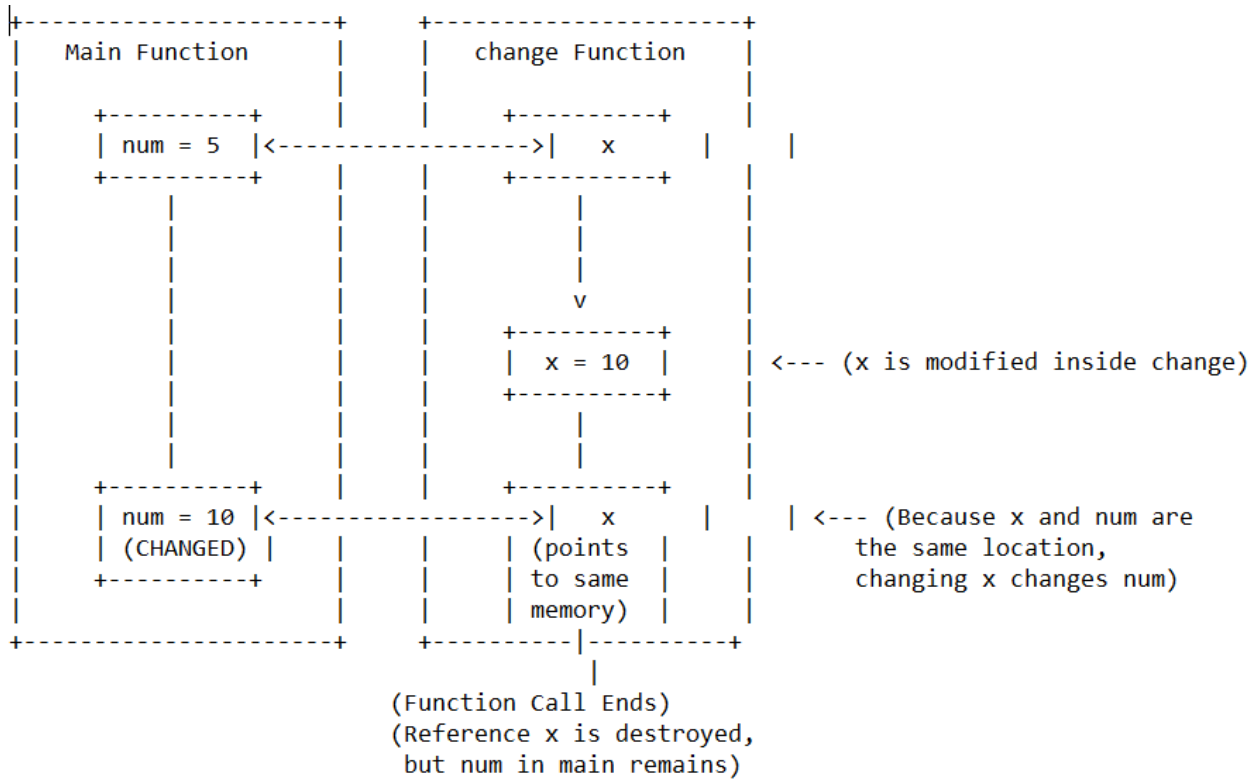
In "Main": Draw a box `num = 5`.

In "change": Draw a box `x` (initially empty).

Show a double-headed arrow or a line connecting `num` and `x` with a label "Same Memory Location".

Inside `change`: Show `x = 10`.

Back in "Main": Show `num` is now `10`



Code Demo (from PDF - modified for clarity):

```
#include <iostream>
void change(int& x) { // Receives a REFERENCE to the original variable
   std::cout << "Inside change(), x was: " << x << std::endl; // Prints 5
   x = 10; // Modifies the ORIGINAL variable
   std::cout << "Inside change(), x is now: " << x << std::endl; // Prints 10
}

int main() {
   int num = 5;
   std::cout << "Before change(), num is: " << num << std::endl; // Prints 5
   change(num); // Pass num BY REFERENCE (note the & in the function definition)
   std::cout << "After change(), num is: " << num << std::endl; // Now prints 10!
   return 0;
}
```

Key Point: "Changes inside the function are reflected outside, in the caller. `num` in `main` is changed."

Professional Note: More efficient than copying, allows functions to modify arguments.

Engagement: "Quick Practice (Verbal): Look at the `Easy 2` swap problem. Why does using `int& a, int& b` make the swap work?" (Answer: Because the function modifies the original variables).

Potential Humor: "Call by Reference is powerful, but it's like giving someone your house keys. They can change anything inside – make sure you trust them!

Call by Pointer

Narrative/Analogy: "Continuing the house analogy: Call by Reference is like giving someone your house keys. Call by Pointer is like giving them the address written on a piece of paper. They still need to go to the address and use the key (the `` operator) to get inside and make changes."

Definition: "When you pass a variable by pointer, you pass the address of the variable to the function. The function uses the `` (dereference) operator to access or modify the value stored at that address."

Visual Diagram 3: Call by Pointer

Draw two columns: "Main Function" and "change Function".

In "Main": Draw a box `num = 5`. Draw another box `&num` (showing its address, e.g., `0x7fff...`).

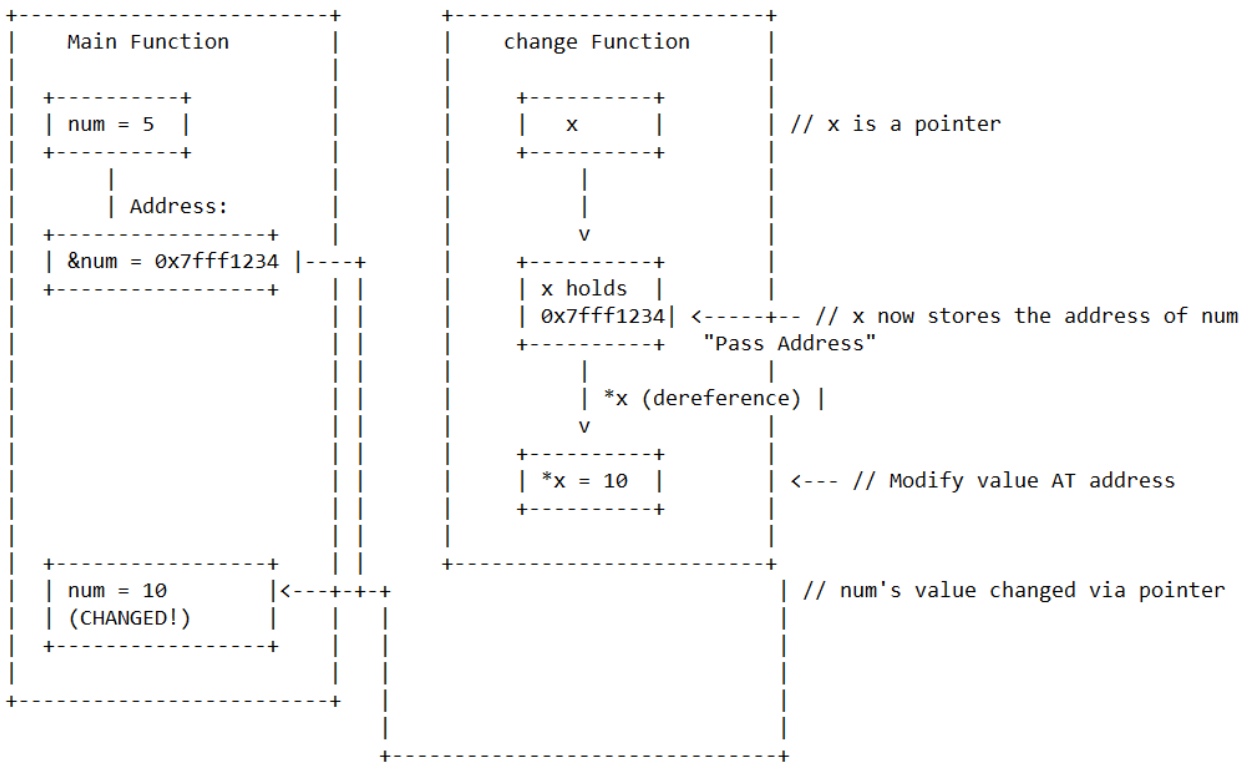
In "change": Draw a box `x` (to hold the address).

Show an arrow from `&num` to `x` with a label "Pass Address".

Inside `change`: Show `x = 10;` (highlight the ``).

Back in "Main": Show `num` is now `10`.

Code Demo (from PDF - modified for clarity):




```
#include <iostream>
void change(int x) { // Receives a POINTER (address)
    if (x != nullptr) { // Always check for null pointer!
        std::cout << "Inside change(), x was: " << x << std::endl; // Prints 5
        x = 10; // Dereference the pointer to modify the ORIGINAL variable
        std::cout << "Inside change(), x is now: " << x << std::endl; // Prints 10
    }
}

int main() {
    int num = 5;
    std::cout << "Before change(), num is: " << num << std::endl; // Prints 5
    change(&num); // Pass the ADDRESS of num (using &)
    std::cout << "After change(), num is: " << num << std::endl; // Now prints 10!
    return 0;
}
```

Key Point: "Like Call by Reference, changes affect the original variable. But you use `` inside the function and `&` when calling it."

Professional Note: Need to handle null pointers. Slightly more syntax (``, `&`) than references.

Engagement: "Medium Problem: Fill in the blank for `doubleValue`. What should the parameter be?" Guide them to `int x`. "Why `x = x  2;`?" (Dereference to get value, multiply, store back).

Comparison: Briefly compare Reference vs Pointer:

References (`&` in param): Easier syntax, no null, must be initialized.

Pointers (`` in param): More flexible (can be reassigned, can be null), requires `` to access value.

Potential Humor: "Call by Pointer is like giving directions. 'Go to 123 Main Street and turn the knob.' You're not handing over the knob itself, just the info on where to find it."

Return by Reference

Narrative/Analogy: "So far, functions have been giving us back copies of values (like photocopies). What if a function could give us back direct access to something it knows about? Like giving you the key to a specific locker in a bank vault, so you can put something in or take something out directly. That's Return by Reference."

Definition: "When a function returns a reference (`&` in the return type), it returns a reference (an alias) to an existing variable, not a copy. This allows the caller to potentially modify the original variable through the returned reference."

Warning: "You can only return a reference to something that will still exist after the function ends. Never return a reference to a local variable!"

Code Demo (from PDF - modified for clarity and safety):

```
#include <iostream>

// SAFE: Returning reference to an element in an array passed in
int& getFirst(int arr[], int size) {
   // Check bounds!
   if (size > 0) {
      return arr[0]; // Return a reference to the first element
   }
   // Handle error case (simplified - ideally throw exception or return a flag)
   // For now, let's assume valid input for the example
   return arr[0]; // This is risky if size <= 0, but demonstrates the concept
}

// UNSAFE EXAMPLE (DO NOT DO THIS):
/
int& getLocalValue() {
   int localVar = 42; // This is destroyed when function ends!
   return localVar;   // Returning reference to destroyed variable - DISASTER!
}
/

int main() {
```

```
    int data[3] = {1, 2, 3};

    // Get a reference to the first element
    int& firstRef = getFirst(data, 3); // firstRef is now an alias for data[0]

    std::cout << "Original first element: " << data[0] << std::endl; // Prints 1

    // Modify the original element THROUGH the reference
    firstRef = 99; // This changes data[0] directly!

    std::cout << "Modified first element: " << data[0] << std::endl; // Prints 99

    // You can also use the return value directly on the left side of assignment!
    getFirst(data, 3) = 150; // Sets data[0] to 150 via the returned reference

    std::cout << "Final first element: " << data[0] << std::endl; // Prints 150

    return 0;
}
```

Key Point: "Returning a reference lets you modify the variable in the caller through the function's return value. `getFirst(data, 3) = 99;` is a powerful idiom."

Professional Note: Extremely useful for operator overloading (e.g., `[]`, `=` operators). Requires careful lifetime management.

Engagement: "Hard Problem: The `largest` function. How can it return a reference to the largest element? What does `largest(data, 5) = 0;` do?" (Answer: It finds the largest element in `data` and sets that specific element to 0).

Potential Humor: "Return by Reference is like the function saying, 'Here's the key to the fridge. Help yourself, but please don't eat everything!' Powerful, but requires trust and care."

Display or draw a quick comparison table:

```
| Method           | How Data is Passed | Changes Affect Original? | Syntax (Func Def) | Syntax (Call) |
|------------------|--------------------|--------------------------|-------------------|---------------|
| Call by Value    | Copy of value      | No                       | `int x`           | `func(var)`   |
| Call by Reference| Direct access      | Yes                      | `int& x`          | `func(var)`   |
| Call by Pointer  | Address of value   | Yes (via x)              | `int x`           | `func(&var)`  |
| Return by Ref    | Returns alias      | Yes (if assigned)        | `int& func(...)`  | `ref = func()`|
```