Berkeley MSSE
MOLECULAR SCIENCE & SOFTWARE ENGINEERING

Lecture 12:

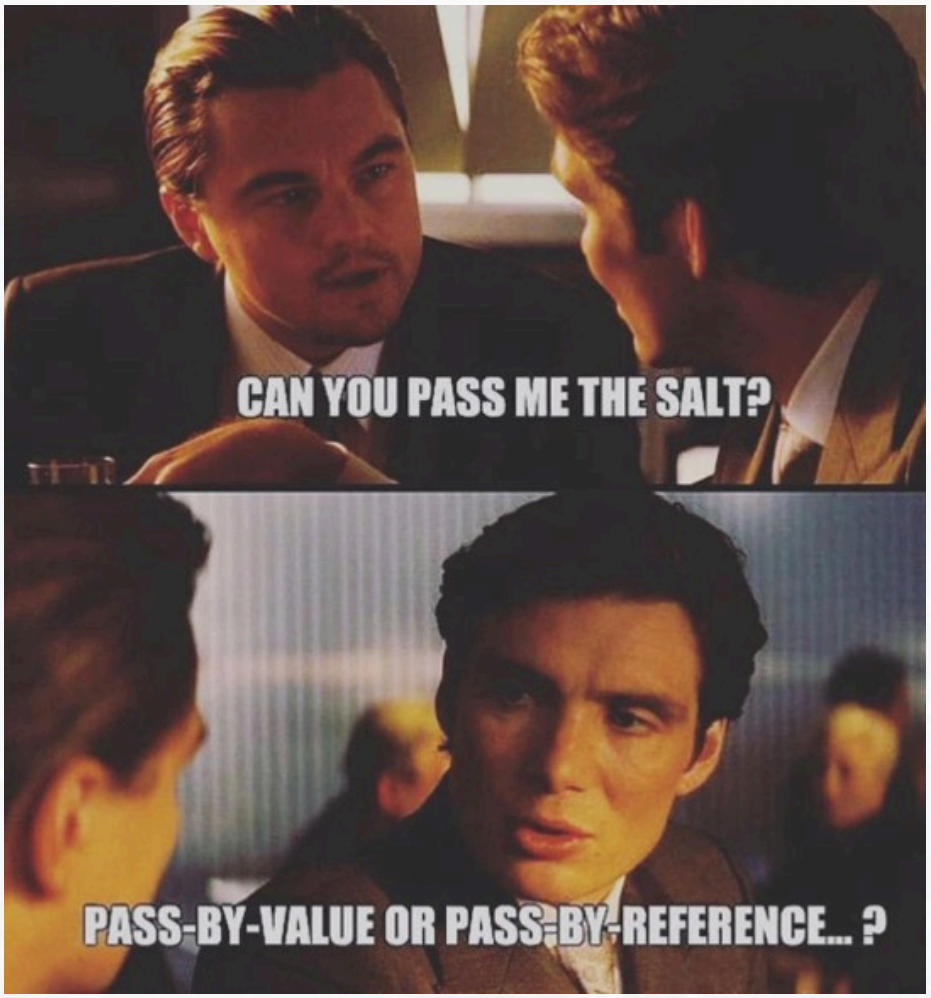
Vectors, Functions, and Multi-file Programs

Jessica Nash

University California, Berkeley

Python for Molecular Sciences

MSSE 272, 3 Units

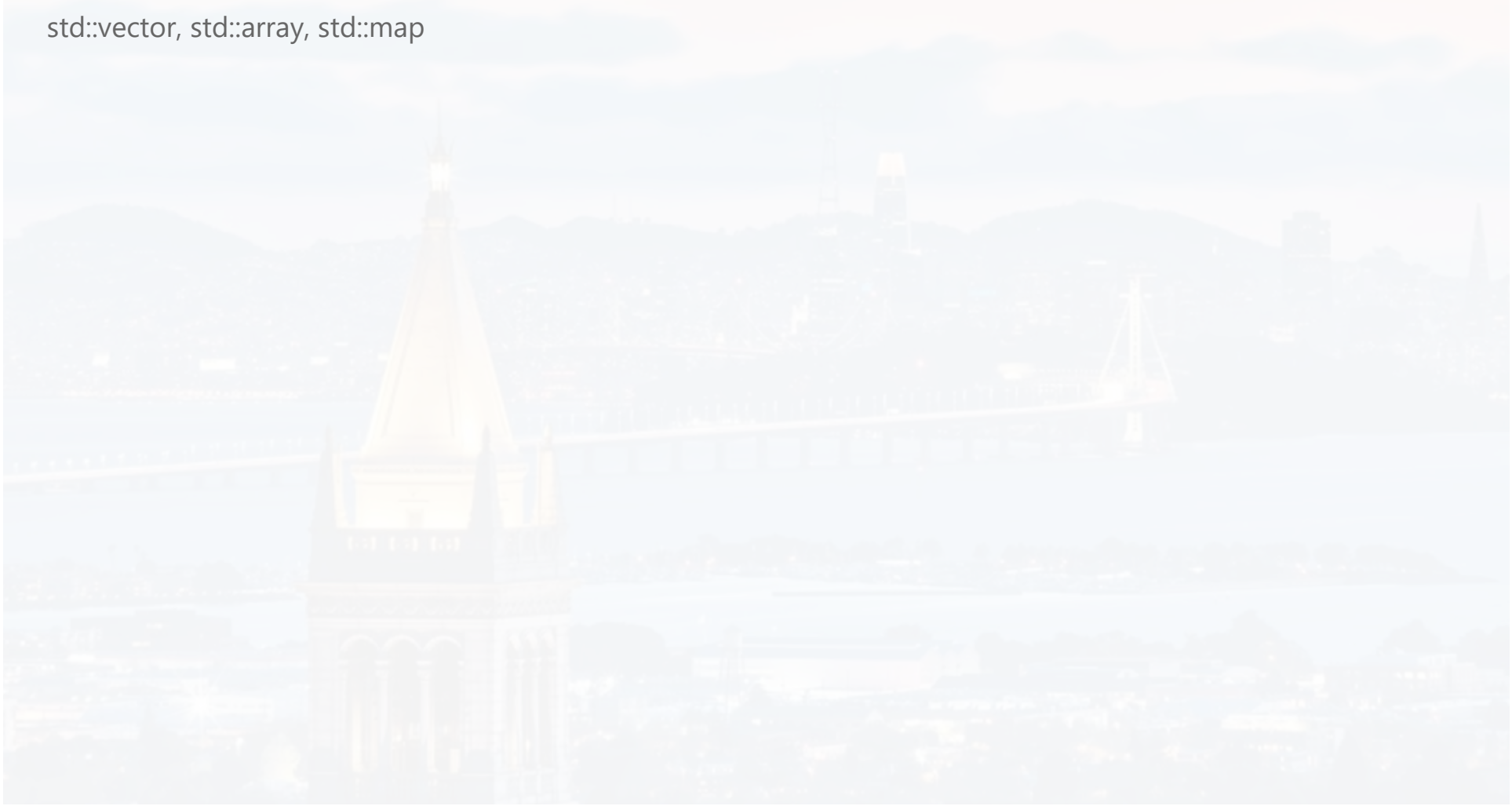


Outline

- Standard Containers
- Argument Passing to Functions in C++
- More Functions: Function overloading and returning multiple values
- C++ Exceptions
- Organizing Code: Multifile C++ Projects

C++ Containers

std::vector, std::array, std::map



Review: C-Style Array

Arrays in C and C++ have these properties:

- Hold multiple values of the same type
- Fixed size determined at compile time
- Access elements by index
- Memory-efficient and fast

```cpp
#include <iostream>

int main() {
    int arr[5] = {10, 20, 30, 40, 50};

    for(int i = 0; i < 5; i++) {
        std::cout << "Element " << i << ": " <<
                     arr[i] << std::endl;
    }

    // arr[10] = 99;  // No bounds checking!

    return 0;
}
```

Key Limitations:

- No bounds checking - can access invalid memory
- Size must be known when writing the code
- No built-in size tracking
- Difficult to resize

std::array Features

std::array wraps C-style arrays with additional safety:

- Same performance as C arrays
- Knows its own size
- Provides bounds checking when requested
- Works with C++ algorithms
- Size still fixed at compile time

Template syntax: std::array<type, size>

```cpp
#include <array>
#include <iostream>

int main() {
    // C-style array
    int c_array[5] = {10, 20, 30, 40, 50};

    // std::array equivalent
    std::array<int, 5> safe_array = {10, 20, 30, 40, 50};

    // Built-in methods
    std::cout << "Size: " << safe_array.size() << std::endl;
    std::cout << "First: " << safe_array[0] << std::endl;

    return 0;
}
```

Access Methods Compared

[] Operator

- Fast access, no overhead
- No bounds checking
- Same behavior as C arrays

.at() Method

- Bounds checking included
- Throws exception if index invalid
- Safer for uncertain indices

Best practice: Use .at() when bounds are uncertain, [] when performance is critical.

```cpp
#include <array>
#include <iostream>

int main() {
    std::array<int, 5> data = {10, 20, 30, 40, 50};

    // Fast access - no bounds checking
    std::cout << data[0] << std::endl;

    // Safe access - bounds checked
    std::cout << data.at(0) << std::endl;

    // This would throw an exception:
    // std::cout << data.at(10) << std::endl;

    return 0;
}
```

When Size Must Be Determined at Runtime

Both C arrays and std::array require compile-time size:

- Reading data from files
- User input determining collection size
- Processing results of unknown quantity
- Building collections incrementally

Problem: How do you store 10 elements? 100? 1000? The answer depends on runtime conditions.

```cpp
#include <iostream>

int main() {
    int count;
    std::cout << "How many numbers? ";
    std::cin >> count;

    // This won't work - size must be compile-time constant
    // std::array<int, count> numbers;

    // This won't work either
    // int arr[count];  // Not standard C++

    // We need a solution for dynamic sizing

    return 0;
}
```

std::vector Solves Dynamic Sizing

std::vector provides arrays that can change size:

- Can start empty and grow as needed
- Automatically manages memory allocation
- Can shrink or grow during execution
- Provides same element access as arrays
- Maintains fast random access

Key capability: Size determined and modified at runtime

```cpp
#include <vector>
#include <iostream>

int main() {
    // Start with empty vector
    std::vector<int> numbers;

    // Add elements as needed
    numbers.push_back(42);
    numbers.push_back(17);
    numbers.push_back(99);

    std::cout << "Size: " << numbers.size() << std::endl;
    std::cout << "First: " << numbers[0] << std::endl;

    return 0;
}
```

Key Vector Methods

- **.size()** - returns current number of elements
- **.push_back(value)** - adds element to end
- **.empty()** - checks if vector has zero elements
- **.front()** - accesses first element
- **.back()** - accesses last element
- **.at(index)** - safe access with bounds checking

Access patterns: Same as std::array - [] for speed, .at() for safety

```cpp
#include <vector>
#include <iostream>

int main() {
    std::vector<int> data = {10, 20, 30};

    // Size and emptiness
    std::cout << "Size: " << data.size() << std::endl;
    std::cout << "Empty? " << data.empty() << std::endl;

    // Element access
    std::cout << "First: " << data.front() << std::endl;
    std::cout << "Last: " << data.back() << std::endl;

    // Safe indexed access
    std::cout << "Middle: " << data.at(1) << std::endl;

    return 0;
}
```

Return type of .size() method

Container .size() methods return size_t, not int:

- size_t is an unsigned integer type
- Designed specifically for object sizes
- Can only represent zero and positive values
- Matches the conceptual nature of container sizes

Why use size_t for loops?

- Type consistency with .size() return value
- Avoids signed/unsigned comparison issues
- Standard practice in professional C++ code
- Some compilers may warn about type mismatches

```cpp
#include <vector>
#include <array>
#include <iostream>

int main() {
    std::vector<int> vec = {10, 20, 30, 40, 50};
    std::array<int, 3> arr = {1, 2, 3};

    // Mixed types: int vs size_t
    for(int i = 0; i < vec.size(); i++) {
        std::cout << vec[i] << " ";
    }
    std::cout << std::endl;

    // Consistent types: both size_t
    for(size_t i = 0; i < arr.size(); i++) {
        std::cout << arr.at(i) << " ";
    }

    return 0;
}
```

std::map Features

std::map stores key-value pairs:

- Similar to Python dictionaries
- Keys must be unique
- Automatically keeps keys sorted
- Fast lookup by key
- Both key and value types specified

Example Mapping:

```
"hydrogen" → 1.008
"carbon" → 12.011
"oxygen" → 15.999
```

```cpp
#include <map>
#include <string>
#include <iostream>

int main() {
    // Map from string to double
    std::map<std::string, double> atomic_weights;

    // Insert key-value pairs
    atomic_weights["hydrogen"] = 1.008;
    atomic_weights["carbon"] = 12.011;
    atomic_weights["oxygen"] = 15.999;

    std::cout << "Map size: " << atomic_weights.size() << std::endl;

    return 0;
}
```

Two Access Methods

[] Operator

- Creates key if it doesn't exist
- Assigns default value to new keys
- Convenient for insertion and known keys

.at() Method

- Throws exception if key doesn't exist
- Safe for accessing uncertain keys
- Never creates new entries

Key difference: [] creates, .at() throws

```cpp
#include <map>
#include <string>
#include <iostream>

int main() {
    std::map<std::string, double> weights;
    weights["hydrogen"] = 1.008;
    weights["carbon"] = 12.011;

    // Access existing key
    std::cout << weights["hydrogen"] << std::endl;

    // Safe access - throws if missing
    std::cout << weights.at("carbon") << std::endl;

    // This creates a new entry!
    std::cout << weights["oxygen"] << std::endl;  // 0.0
    std::cout << "Size now: " << weights.size() << std::endl;  // 3

    return 0;
}
```

Choosing the Right Container

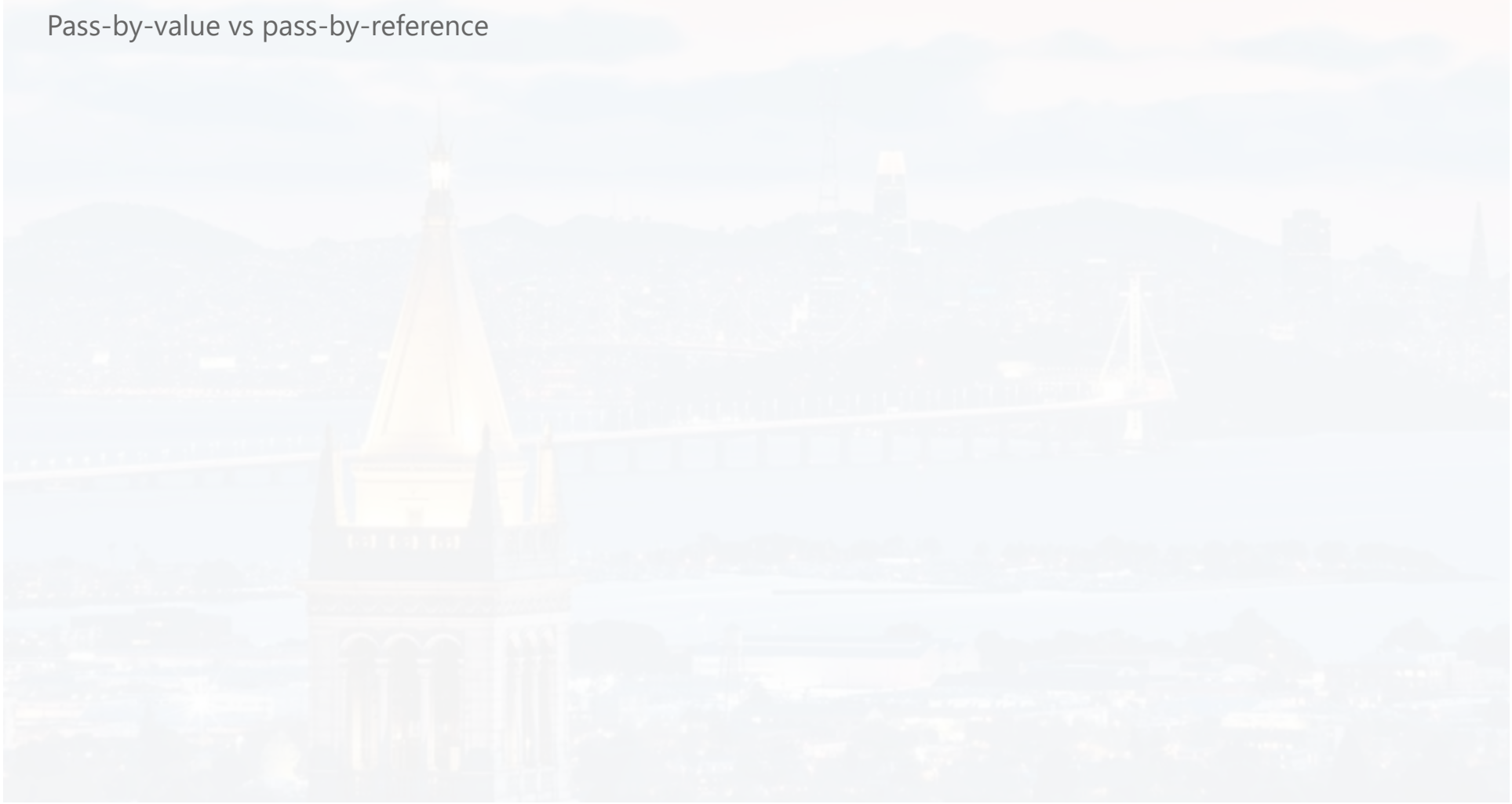
Container	Purpose	Size	When to Use
std::array	Fixed-size collection	Compile-time constant	Coordinates, small fixed datasets
std::vector	Dynamic array	Runtime variable	General-purpose lists
std::map	Key-value storage	Runtime variable	Lookups, associations

Common Safety Features

All containers provide **.size()**, **.empty()**, and safe **.at()** access

Argument Passing to Functions

Pass-by-value vs pass-by-reference



```
double calculate_area(double length, double width) {
    return length * width;
}

int main() {
    double my_length = 12.0;
    double my_width = 8.0;

    double area = calculate_area(my_length, my_width);

    return 0;
}
```

Function Components

- Return type: The type that the function will return
- Function name: The text name that is used to call the function
- Parameters: Comma-separated list of inputs to the function

Today we'll focus on how arguments are passed to functions.

C++'s Default Behavior

By default, C++ uses **pass-by-value**:

- Function receives a **copy** of the argument
- Original variable is **not affected** by changes inside the function
- Safe but uses extra memory

Key Point: The function gets copies of your values, not the original variables.

```cpp
#include <iostream>

void tryToModify(int x) {
    x = 100;  // Only changes the copy
    std::cout << "Inside function: " << x << std::endl;
}

int main() {
    int value = 42;
    std::cout << "Before: " << value << std::endl;

    tryToModify(value);  // Passes a copy

    std::cout << "After: " << value << std::endl;  // Still 42

    return 0;
}
```

Arrays Don't Follow Pass-by-Value

Arrays have special behavior:

- Arrays **decay to pointers** when passed to functions
- Function receives the **address** of the first element
- Changes to array elements **affect the original**
- This is why we pass size separately

> **Important:** Arrays are automatically efficient - no copying occurs.

```cpp
#include <iostream>

void modifyArray(int arr[], int size) {
    arr[0] = 999;  // Modifies original!
}

int main() {
    int numbers[3] = {10, 20, 30};
    std::cout << "Before: " << numbers[0] << std::endl;

    modifyArray(numbers, 3);  // Passes address

    std::cout << "After: " << numbers[0] << std::endl;  // Changed!

    return 0;
}
```

Containers Follow Pass-by-Value

Unlike arrays, std::vector and std::array follow normal pass-by-value rules:

- **Entire container gets copied**
- Can be very expensive for large containers
- Wastes time and memory

Problem: Copying a million-element vector just to read it is wasteful.

```cpp
#include <vector>
#include <iostream>

void printVector(std::vector<int> data) {  // Copies entire vector!
    std::cout << "Size: " << data.size() << std::endl;
}

int main() {
    std::vector<int> huge_data(1000000, 42);  // 1 million elements

    printVector(huge_data);  // Copies 1 million elements!

    return 0;
}
```

We could use pointers to avoid copying

Pointers pass the address, so no copying occurs:

- Function receives the memory address
- No copying of large data structures
- Changes affect the original

Problems with this approach:

- **Must dereference:** `(*data).size()` instead of `data.size()`
- **Function calls:** Must remember & when calling
- **Safety issues:** Pointers can be null or invalid
- **Easy mistakes:** Pointer arithmetic, wrong dereferencing

Result: Pointers work but are error-prone and syntactically clunky for this use case.

```cpp
#include <vector>
#include <iostream>

void printVector(std::vector<int>* data) {
    std::cout << "Size: " << (*data).size() << std::endl;
    std::cout << "First: " << (*data)[0] << std::endl;
}

int main() {
    std::vector<int> huge_data(1000000, 42);

    printVector(&huge_data);  // Must use & explicitly

    return 0;
}
```

Potential Issues:

- `printVector(nullptr)` - crashes program
- `data++` - accidentally moves pointer
- Forget & in function call - won't compile

C++ provides a better solution: references

References give us the efficiency of pointers with safer, cleaner syntax:

How to use references:

Add & to the parameter type:

```
void printVector(std::vector<int>& data)
```

Advantages over pointers:

- **Clean syntax:** `data.size()` (no dereferencing needed)
- **Simple calls:** `printVector(myVector)` (no & needed)
- **Cannot be null:** Must refer to valid object
- **Cannot be reassigned:** Always refers to same object
- **No pointer arithmetic:** Cannot accidentally move reference

Benefits:

- No copying (efficient like pointers)
- Clean syntax (looks like normal variables)
- Cannot be null (safe)
- Simple function calls

The & symbol has two different meanings

You've seen & before with pointers, but this is different:

With pointers:	In parameters:
`&variable` → gets address	`int& x` → creates alias

Key differences:

- **Address operator:** Used in expressions
- **Reference parameter:** Used in declarations
- **Context matters:** Same symbol, different meanings

```cpp
#include <iostream>

// Reference parameter (creates alias)
void printValue(int& x) {
    std::cout << "Value: " << x << std::endl;
}

int main() {
    int number = 42;

    // Address operator (gets memory address)
    int* ptr = &number;

    // Function call with reference
    printValue(number);  // No & needed!

    return 0;
}
```

Summary:

- `&number` → address operator
- `int& x` → reference parameter
- `printValue(number)` → no & in calls!

const Prevents Modification

The const keyword creates variables that cannot be changed:

- Must be **initialized when declared**
- Cannot be modified after initialization
- Compiler enforces this rule

Purpose: Prevents accidental modifications and enables optimizations.

```cpp
#include <iostream>

int main() {
    const int x = 10;  // Must initialize

    std::cout << x << std::endl;  // OK - reading

    x = 20;  // COMPILER ERROR!

    return 0;
}
```

Compiler Error:

```
error: cannot assign to variable 'x'
with const-qualified type 'const int'
```

const and & are often used together for large data structures

Combining const and & gives us efficiency and safety:

const& provides:

- **&:** No copying (efficient)
- **const:** Cannot modify (safe)

Perfect for: Large objects you only need to read from.

> **Best practice:** Use const& for containers you don't need to modify.

```cpp
#include <vector>
#include <iostream>

void printVector(const std::vector<int>& data) {
    std::cout << "Size: " << data.size() << std::endl;
    std::cout << "First: " << data[0] << std::endl;
    // data.push_back(42);  // ERROR! const prevents this
}

int main() {
    std::vector<int> huge_data(1000000, 42);

    printVector(huge_data);  // No copying!

    return 0;
}
```

Argument Passing Summary

Method	Syntax	When to Use	Examples
Pass-by-Value	`int x`	Small, simple types	int, double, char
const Reference	`const vector<int>& v`	Large objects, read-only	vector, string, array
Reference	`vector<int>& v`	When you need to modify	Sorting, input functions
Arrays	`int arr[]`	C-style arrays	Legacy code, C libraries

Most Common Choice

Use **const&** for containers you read from, **regular types** for simple values

Function Overloading

Same name, different parameters

Python Approach

One function name = one function

In Python, if you want different behavior, you typically:

- Use different function names
- Use default parameters
- Check parameter types inside the function

```python
import math

def area_circle(radius):
    return 3.14159 * radius * radius

def area_rectangle(length, width):
    return length * width

def area_triangle(side1, side2, side3):
    s = (side1 + side2 + side3) / 2
    return math.sqrt(s * (s - side1) * (s - side2) * (s - side3))

# Different names for different shapes
print(area_circle(5.0))
print(area_rectangle(4.0, 6.0))
print(area_triangle(3.0, 4.0, 5.0))
```

C++ Approach

One logical name, multiple implementations

C++ allows function overloading:

- Same intuitive name for related operations
- Compiler chooses the right function automatically
- Based on parameter types and count

```cpp
#include <cmath>

double area(double radius) {
    return 3.14159 * radius * radius;
}

double area(double length, double width) {
    return length * width;
}

double area(double side1, double side2, double side3) {
    double s = (side1 + side2 + side3) / 2;
    return sqrt(s * (s - side1) * (s - side2) * (s - side3));
}

// Same name, compiler picks the right one
std::cout << area(5.0) << std::endl;           // Circle
std::cout << area(4.0, 6.0) << std::endl;      // Rectangle
std::cout << area(3.0, 4.0, 5.0) << std::endl; // Triangle
```

Multiple Functions, Same Name

C++ allows multiple functions with the same name if they have different parameters:

- **Different number** of parameters
- **Different types** of parameters
- Compiler chooses the right one automatically

Benefit: One intuitive name for related operations

```cpp
#include <iostream>
#include <cmath>

double area(double radius) {
    return 3.14159 * radius * radius;  // Circle
}

double area(double length, double width) {
    return length * width;  // Rectangle
}

double area(double side1, double side2, double side3) {
    double s = (side1 + side2 + side3) / 2;
    return sqrt(s * (s - side1) * (s - side2) * (s - side3));  // Triangle (Heron's formula)
}

int main() {
    std::cout << area(5.0) << std::endl;           // Circle
    std::cout << area(4.0, 6.0) << std::endl;      // Rectangle
    std::cout << area(3.0, 4.0, 5.0) << std::endl; // Triangle

    return 0;
}
```

What Makes Functions Different?

Functions can be overloaded based on:

✓ Valid Differences

- Number of parameters
- Types of parameters
- Order of parameter types

✗ Not Valid

- Only return type differs
- Only parameter names differ

```
// ✓ Valid overloads
void print(int x);
void print(double x);
void print(int x, int y);

// ✗ Invalid - only return type differs
int calculate(double x);
double calculate(double x);  // ERROR!

// ✗ Invalid - only parameter name differs
void process(int value);
void process(int number);  // ERROR!

// ✓ Valid - different parameter order
void mix(int a, double b);
void mix(double a, int b);
```

Consider This Scenario

We want to create a function called `square` that can:

- Square a single number
- Square every element in a vector

Goal: Use the same function name for both

```
square(5.0) → 25.0
square({1.0, 2.0, 3.0}) → {1.0, 4.0, 9.0}
```

Questions to Consider:

- Is this a good candidate for function overloading?
- Why or why not?
- What makes functions different enough to overload?

Think about the overloading rules we just learned...

What would the functions look like?

Think about:

- What would the parameter types be?
- What would the return types be?
- Are they different enough to overload?
- Does it make logical sense?

Consider your answer, then we'll see the solution...

Why This Works Well

✓ Meets All Requirements

- **Different parameter types:** double vs vector<double>&
- **Same logical operation:** Both square their input
- **Intuitive naming:** "square" makes sense for both

Additional Benefits:

- Compiler chooses the right function automatically
- Natural interface - same operation, different data
- Common pattern in scientific computing

Result: Clean, intuitive code that works with both individual values and collections

```cpp
#include <vector>

double square(double x) {
    return x * x;
}

std::vector<double> square(const std::vector<double>& vec) {
    std::vector<double> result;
    for (double x : vec) {
        result.push_back(x * x);
    }
    return result;
}

int main() {
    double single_result = square(5.0);

    std::vector<double> data = {1.0, 2.0, 3.0};
    std::vector<double> vector_result = square(data);

    return 0;
}
```

Automatic Type Deduction

The `auto` keyword lets the compiler determine variable types automatically:

- Compiler examines the initialization value
- Deduces the appropriate type
- Variable behaves exactly as if you specified the type manually
- Especially useful with complex return types

Key rule: Variable must be initialized when using `auto`

Benefit: Don't need to remember exact return types

```cpp
#include <vector>

// Our overloaded square functions
double square(double x) { return x * x; }
std::vector<double> square(const std::vector<double>& vec) { /* ... */ }

int main() {
    // Explicit types
    double single_result = square(5.0);
    std::vector<double> vector_result = square({1.0, 2.0});

    // Equivalent with auto
    auto single_auto = square(5.0);          // double
    auto vector_auto = square({1.0, 2.0});   // vector<double>

    // Much more convenient!

    return 0;
}
```

Exception Handling

Dealing with runtime errors

Python Exceptions

You're already familiar with Python's approach:

- Automatic exception throwing
- `try/except` blocks
- Built-in exception types

```python
def print_element(numbers, index):
    try:
        print(f"Element {index}: {numbers[index]}")
    except IndexError:
        print(f"Index {index} out of range, printing last element:
{numbers[-1]}")

# Python automatically throws exceptions
numbers = [10, 20, 30]
print_element(numbers, 1)   # Works fine
print_element(numbers, 10)  # IndexError, prints last
```

C++ Exceptions

C++ has similar concepts but different syntax:

- `try/catch` blocks
- Manual exception throwing with `throw`
- Standard library exception types

```cpp
#include <iostream>
#include <vector>

void printElement(const std::vector<int>& numbers, int index) {
    try {
        std::cout << "Element " << index << ": " << numbers.at(index) << std::endl;
    }
    catch (const std::out_of_range& e) {
        std::cout << "Index " << index << " out of range, printing last element: "
                  << numbers.back() << std::endl;
    }
}

std::vector<int> numbers = {10, 20, 30};
printElement(numbers, 1);   // Works fine
printElement(numbers, 10);  // out_of_range, prints last
```

The try-catch Structure

When operations might fail, wrap them in try-catch:

try block

Code that might throw an exception

catch block

Handles the exception if it occurs

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> numbers = {1, 2, 3};

    try {
        // This will automatically throw if index is invalid
        std::cout << numbers.at(10) << std::endl;
    }
    catch (const std::out_of_range& e) {
        std::cout << "Index error: " << e.what() << std::endl;
    }

    return 0;
}
```

The throw Statement

Sometimes you need to throw exceptions manually:

throw statement

Creates and throws an exception when you detect a problem

When to throw manually:

- Validating function parameters
- Checking business rules
- Detecting logical errors

Pattern: Check for problems, then throw appropriate exception type

```cpp
#include <iostream>
#include <stdexcept>

void validateAge(int age) {
    if (age < 0) {
        throw std::invalid_argument("Age cannot be negative");
    }
    if (age > 150) {
        throw std::invalid_argument("Age seems unrealistic");
    }
}

int main() {
    try {
        validateAge(-5);  // This will throw
        std::cout << "Age is valid" << std::endl;
    }
    catch (const std::invalid_argument& e) {
        std::cout << "Error: " << e.what() << std::endl;
    }

    return 0;
}
```

Standard Library Exceptions

A few key exception types you'll commonly use:

`std::runtime_error`
General runtime problems

`std::invalid_argument`
Bad function parameters

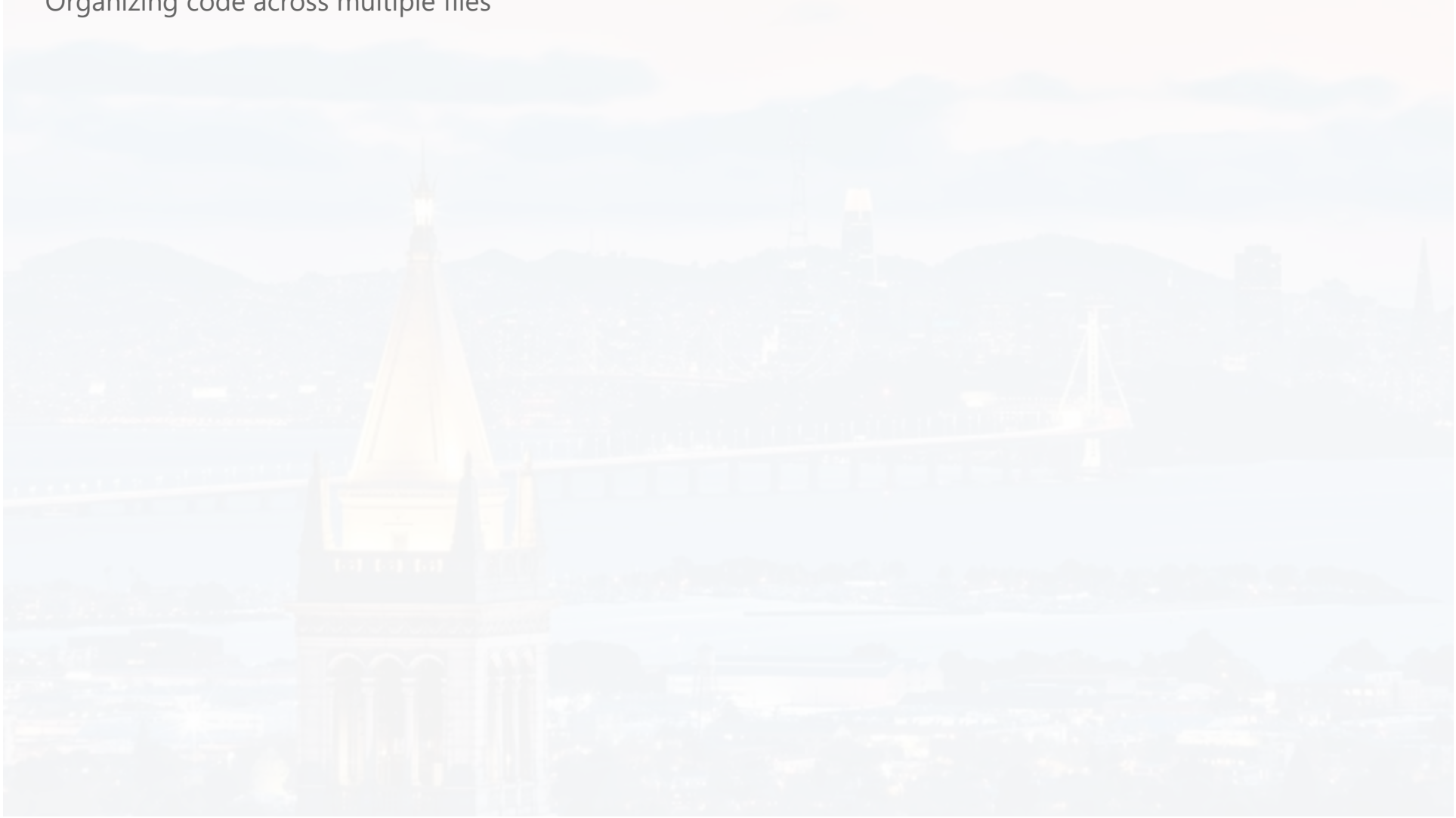
`std::out_of_range`
Array/vector index errors

Remember: Must `#include <stdexcept>` for these types

```cpp
#include <iostream>
#include <stdexcept>
#include <vector>

int main() {
    try {
        // Runtime error example
        throw std::runtime_error("Something went wrong");

        // Invalid argument example
        // throw std::invalid_argument("Bad parameter");

        // Out of range example
        std::vector<int> numbers = {1, 2, 3};
        // std::cout << numbers.at(10);  // Would throw out_of_range

    }
    catch (const std::runtime_error& e) {
        std::cout << "Runtime error: " << e.what() << std::endl;
    }
    catch (const std::invalid_argument& e) {
        std::cout << "Invalid argument: " << e.what() << std::endl;
    }
    catch (const std::out_of_range& e) {
        std::cout << "Out of range: " << e.what() << std::endl;
    }

    return 0;
}
```

Multi-File Projects

Organizing code across multiple files



You wrote a useful function

Your molecular mass calculator works perfectly in one program.

Now you want to use it across multiple lab assignments:

- `stoichiometry.cpp` - reaction calculations
- `solutions.cpp` - concentration analysis
- `empirical.cpp` - formula determination

Question: How do you share the function?

```cpp
#include <iostream>
#include <map>

double calculateMolecularMass(const std::string& formula) {
    std::map<char, double> masses = {
        {'H', 1.008}, {'C', 12.011}, {'N', 14.007},
        {'O', 15.999}, {'S', 32.065}
    };

    double total = 0.0;
    for (char atom : formula) {
        if (masses.count(atom)) {
            total += masses[atom];
        }
    }
    return total;
}

int main() {
    double mass = calculateMolecularMass("H2O");
    std::cout << "Water mass: " << mass << " g/mol" << std::endl;
    return 0;
}
```

Copy the function to each file

Put the entire function in every program that needs it.

Problems:

- Code duplication across multiple files
- Bug fixes must be made in every copy
- Easy to have inconsistent versions
- File sizes grow unnecessarily

In programming, you should avoid repeating the same code in multiple places.

stoichiometry.cpp

```cpp
double calculateMolecularMass(const std::string& formula) {
    // ... entire function copied here ...
}

int main() {
    double mass = calculateMolecularMass("CO2");
    // stoichiometry calculations
}
```

solutions.cpp

```cpp
double calculateMolecularMass(const std::string& formula) {
    // ... same function copied again ...
}

int main() {
    double mass = calculateMolecularMass("NaCl");
    // solution calculations
}
```

Result: If you find a bug in the mass calculation, you must fix it in every file separately.

Include the .cpp file directly

Put the function in `molecular_mass.cpp` and include it.

This seems logical but fails:

- Including .cpp files includes function bodies
- Multiple files = multiple definitions
- Compilation fails with "multiple definition" error

```cpp
// stoichiometry.cpp
#include "molecular_mass.cpp"

int main() {
    double mass = calculateMolecularMass("CO2");
    return 0;
}
```

Compiler Error

```
g++ stoichiometry.cpp solutions.cpp -o program

duplicate symbol 'calculateMolecularMass' in:
  stoichiometry.o
  solutions.o
linker command failed
```

Each .cpp file creates its own copy of the function.

Review: Declaration vs Definition

Variables can be declared separately from their definition:

- **Declaration:** "This variable exists somewhere"
- **Definition:** "Here is the actual variable and value"

Functions work the same way:

- **Declaration:** "This function exists somewhere"
- **Definition:** "Here is the actual function body"

This is called **forward declaration**.

```cpp
#include <iostream>

// Function declaration (forward declaration)
double calculateMolecularMass(const std::string& formula);

int main() {
    double mass = calculateMolecularMass("H2O");
    std::cout << "Water: " << mass << " g/mol" << std::endl;
    return 0;
}

// Function definition (the actual implementation)
double calculateMolecularMass(const std::string& formula) {
    // ... implementation here ...
    return 18.015;  // simplified for example
}
```

Separate declarations from definitions

Header files (.h) contain only function declarations.

Source files (.cpp) contain the function definitions.

Benefits:

- Share declarations without duplicating code
- One definition, multiple uses
- Easy maintenance and updates
- Faster compilation

molecular_mass.h

```cpp
#ifndef MOLECULAR_MASS_H
#define MOLECULAR_MASS_H

#include <string>

// Function declaration only
double calculateMolecularMass(const std::string& formula);

#endif
```

molecular_mass.cpp

```cpp
#include "molecular_mass.h"
#include <unordered_map>

// Function definition
double calculateMolecularMass(const std::string& formula) {
    std::unordered_map<char, double> masses = {
        {'H', 1.008}, {'C', 12.011}, {'N', 14.007},
        {'O', 15.999}, {'S', 32.065}
    };

    double total = 0.0;
    for (char atom : formula) {
        if (masses.count(atom)) {
            total += masses[atom];
        }
    }
    return total;
}
```

Include the header, not the source

Programs include the .h file to access function declarations.

Include syntax:

- `#include "myfile.h"` - Your headers (quotes)
- `#include <iostream>` - System headers (brackets)

Compilation: Link all .cpp files together.

```cpp
// stoichiometry.cpp
#include <iostream>
#include "molecular_mass.h"

int main() {
    double co2_mass = calculateMolecularMass("CO2");
    double water_mass = calculateMolecularMass("H2O");

    double ratio = co2_mass / water_mass;
    std::cout << "CO2/H2O mass ratio: " << ratio << std::endl;

    return 0;
}
```

Compilation Command

```
g++ stoichiometry.cpp molecular_mass.cpp -o program
```

Compiles both .cpp files and links them together.

Problem: Multiple inclusion

Headers can be included multiple times in the same file, causing redefinition errors.

Header guards prevent this:

- `#ifndef SYMBOL` - "If not defined..."
- `#define SYMBOL` - "Define it now"
- `#endif` - "End conditional"

Always use header guards in every .h file.

```cpp
// molecular_mass.h

#ifndef MOLECULAR_MASS_H
#define MOLECULAR_MASS_H

#include <string>

double calculateMolecularMass(const std::string& formula);

#endif  // MOLECULAR_MASS_H
```

How It Works

First inclusion: MOLECULAR_MASS_H undefined → include content
Second inclusion: MOLECULAR_MASS_H defined → skip content

Project Structure

molecular_mass.h

```
#ifndef MOLECULAR_MASS_H
#define MOLECULAR_MASS_H

#include <string>

double calculateMolecularMass(
  const std::string& formula);

#endif
```

molecular_mass.cpp

```
#include "molecular_mass.h"
#include <map>

double calculateMolecularMass(
  const std::string& formula) {
  // ... implementation ...
}
```

stoichiometry.cpp

```
#include <iostream>
#include "molecular_mass.h"

int main() {
  double mass =
calculateMolecularMass("CO2");
  // ... calculations ...
}
```

Compilation

```
g++ stoichiometry.cpp molecular_mass.cpp -o stoichiometry
```

Result: One function definition, usable across multiple programs with no code duplication.

Conclusion

C++ Advanced Concepts Summary

C++ Advanced Concepts

- **Standard Containers:** vector, array, and map for organizing data efficiently
- **Argument Passing:** Pass-by-value vs const& for performance and safety
- **Function Overloading:** Multiple functions with the same name but different parameters
- **Multiple Return Values:** Using references and std::pair to return multiple results
- **Exception Handling:** try/catch blocks for robust error handling
- **Multi-file Projects:** Headers, source files, and organizing large codebases

Key Takeaway

You now have the tools to write efficient, organized, and robust C++ programs for scientific computing.

Next Steps

- Practice with your own molecular science projects
- Apply these concepts to research problems
- Build larger, more complex programs

Thank You for Your Attention