Lecture 9

Function Overloading and Templates

Yohan Jo

Overview

- Function overloading
- Function templates

Function Overloading

Default Arguments in Functions

- You can specify default values for the trailing parameters of a function
- These parameters can be omitted when the function is called

```cpp
#include <iostream>

int divide (int a, int b = 2) {
    int r = a / b;
    return r;
}

int main () {
    std::cout << divide (12) << std::endl;
    // Output: 6
    std::cout << divide (20, 4) << std::endl;
    // Output: 5
    return 0;
}
```

Function Overloading

- Functions can share the same name provided they differ in the type sequence of their parameters
- Cannot overload functions distinguished by return type alone

Function Overloading

```cpp
void mySwapInt(int& x, int& y) {
    int temp = x;
    x = y;
    y = temp;
}

void mySwapDouble(double& x, double& y) {
    double temp = x;
    x = y;
    y = temp;
}

void mySwapChar(char& x, char& y) {
    char temp = x;
    x = y;
    y = temp;
}
```

```cpp
void mySwap(int& x, int& y) {
    int temp = x;
    x = y;
    y = temp;
}

void mySwap(double& x, double& y) {
    double temp = x;
    x = y;
    y = temp;
}

void mySwap(char& x, char& y) {
    char temp = x;
    x = y;
    y = temp;
}
```

Function Overloading

- The compiler uses name mangling to generate unique names for each function version, by including the types and number of parameters and other information (e.g., "`void swap(int& a, int& b);`" → "__Z4swapRiS_")

- For every call to `swap`, the compiler uses the argument type sequence to determine the specific function implementation to invoke

- We can maintain a straightforward, intuitive name reflecting the function's broad functionality, while devising variations to accommodate different basic types

Function Overloading

- But there are still three different implementations for the same functionality

- This makes it difficult to implement and maintain code

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

void mySwap(double& x, double& y) {
    double temp = x;
    x = y;
    y = temp;
}

void mySwap(char& x, char& y) {
    char temp = x;
    x = y;
    y = temp;
}
```

Function Templates

Function Templates

- Function templates enable the creation of functions that can work with any data type

- To declare a function template, use the `template` keyword followed by the template parameters enclosed in angle brackets <>.
    - `template <typename T> function_declaration;`

```
template <typename T>
void mySwap(T &x, T &y) {
    T temp = x;
    x = y;
    y = temp;
}
```

Function Templates

- To call a templated function, provide the function name followed by the template arguments enclosed in angle brackets <> and the function arguments

```cpp
#include <iostream>

int main() {
    int a = 5;
    int b = 10;
    mySwap<int>(a, b);

    std::cout << "a: " << a << std::endl;
    std::cout << "b: " << b << std::endl;
    // a: 10  b: 5

    return 0;
}
```

Template Instantiation

- When a templated function is used with different template arguments, the compiler generates separate code of that function for each unique set of template arguments

```
int main() {
    ...
    mySwap<int>(intA, intB);
    mySwap<double>(doubleX, doubleY);
    ...
}
```

Template Arguments

- Template arguments are **not always required** when invoking a templated function
- The compiler can often **deduce the template arguments** from the provided function arguments

```cpp
#include <iostream>

int main() {
    int a = 5;
    int b = 10;
    mySwap(a, b);

    std::cout << "a: " << a << std::endl;
    std::cout << "b: " << b << std::endl;
    // a: 10  b: 5

    return 0;
}
```

Template Arguments

- Explicit specification of template arguments is necessary in situations where the compiler cannot deduce the template types based on the argument list

```
template <typename T>
T createInstance() {
    T instance;
    return instance;
}

int main() {
    createInstance();
    ...
}
```

```
template <typename T>
void processPointer(T* ptr) {
    // Process pointer...
}

int main() {
    processPointer(nullptr);
    ...
}
```

Template Arguments

- A function template can take multiple template arguments

```
#include <iostream>

template <typename T1, typename T2>
void printPair(T1 a, T2 b) {
    std::cout << "(" << a << ", " << b << ")" << std::endl;
}

int main() {
    printPair(2.5, "orange");

    return 0;
}
```

Template Arguments

```
auto it = std::find(vec.begin(), vec.end(), 3);
```

std::find, std::find_if, std::find_if_not

Defined in header `<algorithm>`

```
template< class InputIt, class T >
InputIt find( InputIt first, InputIt last, const T& value );
```

https://en.cppreference.com/w/cpp/algorithm/find

Non-Type Template Arguments

- Non-type template arguments allow you to pass values (not types) as arguments to templates
- Non-type template arguments must be compile-time constants since templates are instantiated at compile-time

```cpp
#include <iostream>

// Compute base^N with loop unrolling
template <int N>
int power(int base) {
    int result = 1;
    for (int i = 0; i < N; ++i)
        result *= base;
    return result;
}

int main() {
    std::cout << "2^3 = " << power<3>(2) << '\n';
}
```

Non-Type Template Arguments

- **Advantage** of using a template argument vs. a regular function argument
  - Since the value is known at compile time, the compiler has more opportunities to optimize the generated code of the function and capture potential errors
- **Disadvantage** of using a template argument vs. a regular function argument
  - Non-type template arguments cannot be used if their values are not known at compile time
  - Each argument value results in a separate instantiation of the template, which can increase the size of the compiled binary

Function Overloading and Function Templates

Original Functions

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

void mySwapDouble(double& x,
double& y) {
    double temp = x;
    x = y;
    y = temp;
}

...
```

Function Overloading

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

void mySwap(double& x,
double& y) {
    double temp = x;
    x = y;
    y = temp;
}

...
```

Function Templates

```
template <typename T>
void mySwap(T &x, T &y) {
    T temp = x;
    x = y;
    y = temp;
}
```

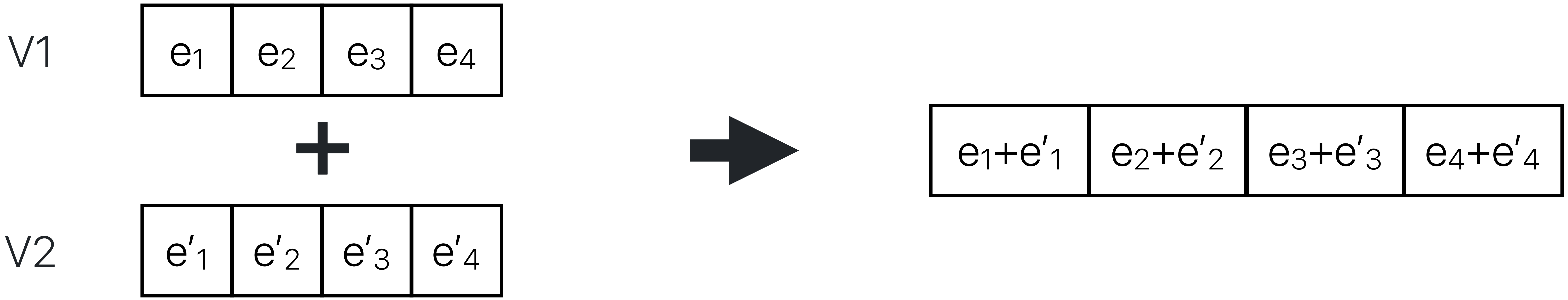
Exercises

Overview

- The goal is to understand how to create and use function templates that can operate on different data types
- We will implement basic vector calculations using function templates
- Implement the `addVectors` and `dotProduct` functions

addVectors

- The `addVectors` function performs element-wise addition of two vectors
- Be careful with the sizes of two vectors!



dotProduct

- The dot product is the sum of the products of corresponding vector components
- It returns a single scalar
- The two vectors must have the same size

V1	e_1	e_2	e_3	e_4

$\bullet$

V2	e'_1	e'_2	e'_3	e'_4

$$\rightarrow \quad e_1 * e'_1 + e_2 * e'_2 + e_3 * e'_3 + e_4 * e'_4$$

Instructions

- Find the TODO sections in `exercise.h` and `test_cases.cpp` files and implement them correctly
- There are a total of 7 TODOs
- $ g++ test_cases.cpp -o run_tests -std=c++17
- $ ./run_tests

Solution

TODO 1 `addVectors`

```
template<typename T>
vector<T> addVectors(const vector<T>& v1, const vector<T>& v2) {
    if (v1.size() != v2.size()) {
        throw invalid_argument("Vectors must have the same size.");
    }
    vector<T> result(v1.size());
    for (size_t i = 0; i < v1.size(); ++i) {
        result[i] = v1[i] + v2[i];
    }
    return result;
}
```

Solution

TODO 2 `dotProduct`

```
template<typename T>
T dotProduct(const vector<T>& v1, const vector<T>& v2) {
    if (v1.size() != v2.size()) {
        throw invalid_argument("Vectors must have the same size.");
    }
    T result = 0;
    for (size_t i = 0; i < v1.size(); ++i) {
        result += v1[i] * v2[i];
    }
    return result;
}
```

Solution

TODO 3

```
vector<T> result = addVectors(v1, v2);
```

TODO 4

```
cout << "Result of `dotProduct`: " << dotProduct(v1, v2) << endl;
```

Solution

TODO 5

```
processVectors<int>();
```

TODO 6

```
processVectors<float>();
```

TODO 7

```
processVectors<double>();
```

Thank you