Function Pointers & Lambdas

Function pointers are essential in C and C++ programming.

Function Pointers in C

```
#include <stdio.h>
// Function prototypes for the operations
int add(int a, int b) {
    return a + b;
}
int subtract(int a, int b) {
    return a - b;
}
int main() {
    // Define function pointers for the operations
    int (*operation)(int, int);
    int x = 10;
    int y = 5;
    int result;
    // Use the function pointer to point to the add function
    operation = add;
    result = operation(x, y);
    printf("Adding: %d + %d = %d\n", x, y, result);
    // Change the function pointer to point to the subtract function
    operation = subtract;
    result = operation(x, y);
    printf("Subtracting: %d - %d = %d\n", x, y, result);
    return 0;}
```

We can make the code more intuitive by using a "typedef".

```
```

```c
#include <stdio.h>
// Function prototypes for the operations
int add(int a, int b) {
    return a + b;
}
int subtract(int a, int b) {
    return a - b;
}
// Using typedef to simplify function pointer declarations
typedef int (*function_pointer_for_operation)(int, int);
int main() {
    // Declare a function pointer using the typedef
    function_pointer_for_operation operation;
    int x = 10;
    int y = 5;
    int result;
    // Use the function pointer to point to the add function
    operation = add;
    result = operation(x, y);
    printf("Adding: %d + %d = %d\n", x, y, result);
    // Change the function pointer to point to the subtract function
    operation = subtract;
    result = operation(x, y);
    printf("Subtracting: %d - %d = %d\n", x, y, result);
    return 0;
}
```

Function Pointers in C++

```cpp
#include <iostream>
#include <functional>
// Function prototypes for the operations
int add(int a, int b) {
    return a + b;
}
```

```
int subtract(int a, int b) {
    return a - b;
}
int main() {
    // Using std::function to declare function pointers
    std::function<int(int, int)> operation;
    int x = 10;
    int y = 5;
    int result;
    // Assign the add function to the operation std::function
    operation = add;
    result = operation(x, y);
    std::cout << "Adding: " << x << " + " << y << " = " << result << std::endl;
    // Change the operation to subtract
    operation = subtract;
    result = operation(x, y);
    std::cout << "Subtracting: " << x << " - " << y << " = " << result << std::endl;
    return 0;
}
```

Similar to C, we can improve our code by using a typedef:

```
#include <iostream>
#include <functional>
// Define the function type using typedef for clarity and reusability
typedef std::function<int(int, int)> MathOperation;
// Function prototypes for the operations
int add(int a, int b) {
    return a + b;
}
int subtract(int a, int b) {
    return a - b;
}
int main() {
    // Declare a variable of type MathOperation
```

```
    MathOperation operation;
    int x = 10;
    int y = 5;
    int result;
    // Assign the add function to the operation
    operation = add;
    result = operation(x, y);
    std::cout << "Adding: " << x << " + " << y << " = " << result << std::endl;
    // Change the operation to subtract
    operation = subtract;
    result = operation(x, y);
    std::cout << "Subtracting: " << x << " - " << y << " = " << result << std::endl;
    return 0;
}
```

Example 1

```
#include <iostream>
#include <vector>
#include <algorithm>
// Comparison functions:
bool ascending_compare(int a, int b) {
    return a < b; // Change this to a > b for descending order
}
bool descending_compare(int a, int b) {
    return a > b; // Change this to a > b for descending order
}
int main() {
    // Create a vector of integers
    std::vector<int> vec = {-8, 5, 2, 9, 1, 5, 6, 3};
    // Sort the vector using the comparison function
    std::sort(vec.begin(), vec.end(), descending_compare);
    // Print the sorted vector
    std::cout << "Sorted vector: ";
```

```
    for (int value : vec) {
        std::cout << value << " ";
    }
    std::cout << std::endl;
    return 0;}
```

Example 2

```
#include <iostream>
#include <chrono>
#include <functional>
#include <thread>
class Timer {
public:
    void callback_after_delay(std::function<void()> callback, int delay) {
        std::this_thread::sleep_for(std::chrono::milliseconds(delay));
        callback();
    }
};
void function_to_callback() {
  std::cout << "Callback invoked!" << std::endl;
}
int main() {
    Timer t;
    t.callback_after_delay(function_to_callback, 3000);
    return 0;}
```

Lambda

Before we learn what is a lambda, let us consider naive code that achieves similar functionality of a lambda but it uses function pointer:

```
#include <algorithm>
#include <iostream>
```

```
#include <vector>
// A simple comparison function
bool compare(int a, int b) {
    return a < b;
}
int main() {
    std::vector<int> data = {5, 3, 9, 1, 6};
    // Sort using a function pointer
    std::sort(data.begin(), data.end(), compare);
    for (int n : data) {
        std::cout << n << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

In super naive terms, a lambda is a function pointer. Just remember this syntax: `[] () {}`

```
#include <algorithm>
#include <iostream>
#include <vector>
int main() {
    std::vector<int> data = {5, 3, 9, 1, 6};
    // Sort using a lambda expression
    auto compare_function_lambda = [](int a, int b) {return a < b; };
    std::sort(data.begin(), data.end(), compare_function_lambda);
    for (int n : data) {
        std::cout << n << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

Revision #4
Created 1 year ago by Preet Kang

Updated 1 year ago by Preet Kang