**Data Structures and Algorithms**

**Chapter 3.0: Strings, Templates, Vectors**

## std::string

Unlike other languages, in C++, string is not part of the built-in data types.

**std::string** is a class in the C++ Standard Library that represents and manipulates sequences of characters. It is defined in the <string> header file and is part of the std namespace. std::string is designed to make working with text in C++ much easier and safer than using C-style character arrays (char * or char[]).

In string form, numbers are treated as text, not as numbers, and thus they cannot be manipulated as numbers (e.g. you can't multiply them). C++ will not automatically convert strings to integer or floating point values or vice-versa (though there are ways to do so).

Unlike C-style strings, which are fixed-size arrays of characters, std::string can change size dynamically to accommodate the text it stores. This means you don't need to worry about buffer overflow issues that can occur with C-style strings.

Example:

```
#include <iostream>
#include <string>
int main() {
        std::string str = "Hello, World!";
        std::cout << str << '\n';
        return 0;
}
```

When using std::string, std::cin will not work as expected because cin breaks on whitespace.

Example:

```
#include <iostream>
#include <string>
int main() {
        std::string name;
        std::cout << "Enter your full name: ";
        std::cin >> name; // Using cin to read a string
        std::cout << "Your full name is " << name << '\n';
        return 0;
}
```

Example output:

```
Enter your full name: John Doe
Your full name is John
```

If the user enters a full name with spaces, `std::cin` will only read the first word and leave the rest in the **input buffer**. This can lead to unexpected behavior because the input buffer will still contain the remaining characters, which may affect subsequent input operations.

Example:

```
#include <iostream>
#include <string>
int main() {
        std::string name;
        std::cout << "Enter your full name: ";
        std::cin >> name; // Using cin to read a string
        std::string address;
        std::cout << "Where do you live?";
        std::cin >> address;
        std::cout << "Your full name is " << name << '\n';
        std::cout << "You live in " << address;
        return 0;
}
```

Example output:

```
Enter your full name: John Doe
Where do you live?Your full name is John
You live in Doe
```

That's definitely not the behavior we want.

What can we do?

To handle this situation, you can use `std::getline()` instead of std::cin to read a full line of input, including spaces.

```
#include <iostream>
#include <string>
int main() {
        std::string name;
        std::cout << "Enter your full name: ";
        std::getline(std::cin, name); // Using getline to read a full line
        std::cout << "Hello, " << name << '\n';
        return 0;
}
```

In this example, std::getline is used to read the entire line of input, including any spaces. This ensures that the entire name entered by the user is captured and stored in the name variable.

Example output, with leading space before the name:

```
Enter your full name:   John Doe
Hello,    John Doe
```

Notice the space before the name. If you don't want to include the space before the string, we can do something better:

```cpp
#include <iostream>
#include <string>
int main() {
        std::string name;
        std::cout << "Enter your full name: ";
        std::getline(std::cin >> std::ws, name); // Using std::ws with getline t
o handle leading whitespace
                std::cout << "Hello, " << name << '\n';
        return 0;
}
```

Example output, with leading space before the name:

```
Enter your full name:    John Doe
Hello, John Doe
```

In the above example, std::ws is used directly within the std::getline function to discard any leading whitespace before reading the user's input into the name variable. This ensures that any leading whitespace characters are ignored before capturing the user's input.

## Templates

What is a template in C++?

The template system in C++ was created to make it easier to create functions (or classes) that can operate on various data types.

Instead of manually creating a lot of mostly identical functions or classes (one for each set of different types), we instead create a single template.

A template describes what a function or class looks like. Unlike a normal definition, in a template we can use one or more placeholder types.

A placeholder type represents some type that is not known at the time the template is written, but that will be provided later.

Once a template is defined, the compiler can use the template to generate as many overloaded functions (or classes) as needed, each using different actual types.

The end result is the same -- we end up with a bunch of mostly-identical functions or classes (one for each set of different types). But we only have to create and maintain a single template, and the compiler does all the hard work for us.

Template parameter declaration:

```cpp
template <typename T>
```

**Function Templates**

• A function template is a function-like definition that is used to generate one or more overloaded functions, each with a different set of actual types. This is what will allow us to create functions that can work with many different types.

• When we create our function template, we use placeholder types (also called type template parameters, or informally template types) for any parameter types, return types, or types used in the function body that we want to be specified later.

Example:

```
#include <iostream>
// Template function to return the maximum of two values
template <typename T>
T max(T a, T b) {
            return (a > b) ? a : b;
}
int main() {
            // Using max with int
            int i1 = 3, i2 = 5;
            std::cout << "Max of " << i1 << " and " << i2 << " is " << max(i1, i2) << '\n';
            // Using max with double
            double d1 = 6.5, d2 = 2.5;
            std::cout << "Max of " << d1 << " and " << d2 << " is " << max(d1, d2) << '\n';
            // Using max with char
            char c1 = 'a', c2 = 'z';
            std::cout << "Max of " << c1 << " and " << c2 << " is " << max(c1, c2) << '\n';
            return 0;
}
```

This allows us to create a generic max function that will work with any data type, rather than being limited to just ints or doubles.

The T is a placeholder representing whatever data type will be used when calling max. When max is called, the T gets replaced with a specific type:

```
int x = 5, y = 10;
int max_int = max(x, y);
```

Here T is replaced with int, so max will compare two ints.

We can also call max with other types, and it will adjust accordingly:

```
double m = 1.5, n = 2.5;
double max_double = max(m, n);
```

Now T is double, so max will compare two doubles.

The key benefit is that we can write the logic for finding the max just once in a generic way, and rely on templates to apply it to whatever types we need.

**Class Templates**

A class template is a template definition for instantiating class types.

Example:

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

template<typename T>
class SinglyLinkedList {
private:
    // Node struct defined inside the SinglyLinkedList class
    struct Node {
        T data;
        Node* next;

        // Constructor
        Node(T value) : data(value), next(nullptr) {}
    };

    Node* head;

public:
    // Constructor
    SinglyLinkedList() : head(nullptr) {}

    // Insert node at head
    void prepend(T value) {
        Node* newNode = new Node(value);
        newNode->next = head;
        head = newNode;
    }

    // Insert node at tail
    void append(T value) {
        Node* newNode = new Node(value);
        if (!head) {
            head = newNode;
            return;
        }
        Node* curr = head;
        while (curr->next) {
            curr = curr->next;
        }
        curr->next = newNode;
    }

    // Print list
    void printList() const {
        Node* curr = head;
        while (curr) {
            std::cout << curr->data << " ";
            curr = curr->next;
        }
```

```
        std::cout << '\n';
    }
};

int main() {
    SinglyLinkedList<int> list1;
    SinglyLinkedList<double> list2;

    list1.prepend(5);
    list1.prepend(10);
    list2.prepend(12.34);
    list2.prepend(56.78);

    list1.printList(); // Output: 10 5
    list2.printList(); // Output: 56.78 12.34

    list1.append(15);
    list2.append(90.12);

    list1.printList(); // Output: 10 5 15
    list2.printList(); // Output: 56.78 12.34 90.12

    return 0;
}
```

The SinglyLinkedList class is declared as templates using this syntax:

```
template<typename T>
class SinglyLinkedList {
        //...
};
```

This allows us to define the classes generically by representing the data type as T instead of a specific type like int or string.

When we use these classes, we instantiate them by providing a specific type:

```
SinglyLinkedList<int> list1;
SinglyLinkedList<double> list2;
```

The benefit of templates here is that we can define these classes generically without needing to know exactly what data type will be used. This allows maximum code reuse and avoids needing to rewrite the linked list logic for each data type we want to use. We get type safety and custom behavior for each data type, but with the simplicity and flexibility of generic code. This is the power of templates in C++!

## std::pair

std::pair in C++ is a template class that provides a way to store a pair of elements/objects as a single unit.

• It is defined in the <utility> header file.

• It can store two elements of either the same or different data types.

The data types are specified as template parameters

Here is the basic syntax for creating a std::pair:

```
std::pair<int, double> p1; //int and double
std::pair<std::string, std::string> p2; //two strings
```

Elements are accessed using the public members first and second:

```
p1.first = 5;
p1.second = 2.5;
p2.first = "Hello";
p2.second = "World";
```

In summary, std::pair allows grouping together a pair of values as a single object, providing convenient access to the elements and overloads to compare pairs. This avoids having to create a separate class for each pair type.

**std::vector**

std::vector is a container class in C++ that represents a dynamic array. It is part of the Standard Library and provides a flexible and efficient way to store and manipulate a sequence of elements. std::vector is defined in the <vector> header as a class template, with a template type parameter that defines the type of the elements.

• Unlike traditional arrays, std::vector can dynamically change its size at runtime to accommodate the number of elements it holds.

• Elements in a std::vector are stored in a contiguous memory block, allowing for fast sequential

access.

• std::vector can automatically resize itself when elements are added or removed, managing memory allocation and reallocation behind the scenes.

• It provides a range of member functions for adding, removing, and accessing elements, as well as utilities for sorting, searching, and iterating over its contents.

Example:

```cpp
#include <iostream>
#include <vector>
int main() {
        std::vector<int> numbers; // Create an empty vector of integers
        // Add elements to the vector using push_back
        numbers.push_back(10);
        numbers.push_back(20);
        numbers.push_back(30);
        // Access elements using index
        std::cout << "Elements in the vector: ";
        for (int num : numbers) {
                std::cout << num << " ";
        }
        std::cout << '\n';
        // Remove the last element from the vector using pop_back
        numbers.pop_back();
        // Access elements after pop_back
        std::cout << "Elements after pop_back: ";
        for (int num : numbers) {
                std::cout << num << " ";
        }
        std::cout << '\n';
        return 0;
}
```