DAY 07

C++ Templates

Templates :

- Templates are a powerful feature of C++ that allow you to write generic code that can work with different types of data. In C++, templates can be used to define functions, classes, and even entire libraries that can work with multiple types of data, without having to write separate code for each type.
- A template is a pattern for creating a generic function or class, where one or more parameters are defined as type placeholders. The type placeholders are then replaced with the actual data types when the template is instantiated.
- The syntax of templates in C++ involves using the template keyword followed by one or more template parameters enclosed in angle brackets <>, which can be used to define generic functions and classes.

```
template <typename T>
void swap(T& a, T& b) {
  T temp = a;
  a = b;
  b = temp;
}
```

Template Function :

- A template function is a function in C++ that can work with multiple data types. The template function allows you to define a generic function that can be used with any data type, without having to write separate code for each type.

```
template <typename T>
T myFunction(T arg1, T arg2) {
  // Function body
}
```

- In this code, template <typename T> declares the function as a template function, and typename T specifies a placeholder for the actual data type that will be used when the function is called. The function myFunction takes two arguments of type T and returns a value of type T.

```
template <typename T>
T max(T a, T b) {
  return (a > b) ? a : b;
}
```

```
int x = 10, y = 20;
std::cout << max(x, y) << std::endl;

double a = 3.14, b = 2.71;
std::cout << max(a, b) << std::endl;
```

Templates with Multiple Parameters :

- Templates with multiple parameters are used in C++ to define generic functions or classes that can work with multiple data types, or with multiple values of different types.
  - Here is an example of a template function with multiple parameters that calculates the sum of two values of different types:

```
template <typename T1, typename T2>
T1 add(T1 a, T2 b) {
  return a + b;
}
```

```
int x = 10;
double y = 3.14;

std::cout << add<int, double>(x, y) << std::endl;
```

  - You can also define a class with multiple template parameters, as shown in the following example:

```
template <typename T1, typename T2>
class MyClass {
  private:
    T1 data1;
    T2 data2;
  public:
    MyClass(T1 d1, T2 d2) : data1(d1), data2(d2) {}
    void printData() {
      std::cout << "Data1: " << data1 << std::endl;
      std::cout << "Data2: " << data2 << std::endl;
    }
};
```

Class template :

- Class templates are a powerful feature of C++ that allow you to define a generic class that can work with different data types. A class template defines a blueprint for a family of classes, where each member of the family is a class that is instantiated with a specific set of template arguments.
  - Here is the syntax for defining a class template:

```
template <typename T>
class MyClass {
  private:
    T data;
  public:
    MyClass(T d) : data(d) {}
    T getData() {
      return data;
    }
    void print();
};
template<class T>
void MyClass<T>::print(){
std::cout << " Hello " << std::endl;
}
```

```
MyClass<int> obj1(10);
std::cout << obj1.getData() << std::endl;

MyClass<std::string> obj2("Hello");
std::cout << obj2.getData() << std::endl;
```

  - You can also define a class template with multiple template parameters, as shown in the following example:

```
template <typename T1, typename T2>
class MyPair {
  private:
    T1 first;
    T2 second;
  public:
    MyPair(T1 f, T2 s) : first(f), second(s) {}
    T1 getFirst() {
      return first;
    }
    T2 getSecond() {
      return second;
    }
};
```

```
MyPair<int, double> p1(10, 3.14);
std::cout << p1.getFirst() << " " << p1.getSecond() << std::endl;

MyPair<std::string, char> p2("Hello", 'W');
std::cout << p2.getFirst() << " " << p2.getSecond() << std::endl;
```

Template specialization :

- Template specialization is a feature of C++ templates that allows you to provide a specialized implementation of a template for a specific data type or set of data types. Template specialization is useful when you need to handle a particular case differently from the general case.
  - Here is the syntax for defining a template specialization:

```
// generic template
template <typename T>
class MyClass {
  // general implementation
};

// specialization for int
template <>
class MyClass<int> {
  // specialized implementation for int
};
```

  - Here is an example of using template specialization to provide a different implementation for a class when it is instantiated with a char* data type:

```
// generic template
template <typename T>
class MyClass {
  public:
    void print(T data) {
      std::cout << "Generic implementation: " << data << std::endl;
    }
};

// specialization for char*
template <>
class MyClass<char*> {
  public:
    void print(char* data) {
      std::cout << "Specialized implementation: " << data << std::endl;
    }
};

int main() {
  MyClass<int> obj1;
  obj1.print(10);

  MyClass<char*> obj2;
  obj2.print("Hello");

  return 0;
}
```