DISCUSSION SESSION

WEEK 8

C++ TEMPLATING & CLASSES

What's the Output?

```
void func(int a, int b) {
        std::cout << a + b << std::endl;
}
int main( ) {
        std::string city = "Las Vegas";
        std::string state = " NV";
        int x = 5, y = 10;
        double m = 2.3, n = 3.2;
        func(x, y);
        func(m, n);
        func(city, state);   }
```

Easy Fix ✅

```
int main( ) {
std::string city = "Las Vegas";
std::string state = " NV";
int x = 5, y = 10;
double m = 2.3, n = 3.2;
func(x, y);
func(m, n);
func(city, state);   }
```

```
  void func(int a, int b) {
        std::cout << a + b << std::endl;
  }
  void func(double a, double b) {
        std::cout << a + b << std::endl;
  }
void func(std::string a, std::string b) {
      std::cout << a + b << std::endl;
 }
```

TEMPLATE PROGRAMMING

A template is a simple yet powerful tool in C++ that allows you to write generic code that works with different data types without sacrificing type safety.

Templates are expanded at compile-time. The compiler does type-checking and then generates specific instances of the code for each type used.

```
template <typename T>
void func(T a, T b) {
        std::cout << a + b;
}
int main( ) {
    int x = 5, y = 10;
    double m = 2.3, n = 3.2;
    std::string city = "Las Vegas", state = " NV";
    func(x, y);
    func(city, state);
    func(m, n);
```

// Compiler internally generates below code for each type used

```
void func(int a, int b) {
    std::cout << a + b;
}
```

```
void func(std::string a, std::string b) {
    std::cout << a + b;
}
```

```
void func(double a, double b) {
    std::cout << a + b;
}
```

```
template <typename T>
void func(T a, T b) {
        std::cout << a + b;
}
int main( ) {
    double x = 5.3, y = 10.7;
    func<double>(x, y); // explicit template instantiation
    func(x, y);  // implicit template instantiation
}
```

Exercise (3 minutes)

Create a template function that takes in two parameters and returns the maximum of both.

In main, test your template function with different types of arguments. Use both implicit and explicit template instantiation.

C++ CLASSES

A class is a user-defined type that serves as a blueprint for creating instances (AKA objects). Classes are a fundamental part of object-oriented programming (OOP), which aims to organize code in a way that models real-world entities.

Key Concepts of Classes

- ✓ Data Members: These are variables that hold the data associated with an object. They define the attributes of the class.
- ✓ Member functions (AKA Methods): These are functions defined inside a class that operate on the data members. They define the behavior of the class.

Key Concepts of Classes (Contd.)

✓ Access Specifiers:

1. Private: Members are accessible **only** within the class.
2. Public: Members are accessible from outside the class.
3. Protected: Members are accessible within the class and by derived classes.

```
class Person {
    public:
        std::string name;
        int age;
        std::string address;
};
int main( ) {
    Person p1, p2; // p1 & p2 are objects of Person class
    p1.name = "John Doe";   p2.name = "Jane Doe";
    p1.age = 25;     p2.age = 24;
    p1.address = "45 Upper College Rd, Kingston, RI 02881";
```

Note: Class declarations end in semicolons!

Note: We can access variables “name,” “age,” and “address” in main because they have **public** access.

How do we initialize/access private data members then?

Setters and Getters!

```
class Person {
    private:
        std::string name;
         int age;
    public:
        void setAge(int userAge); // a setter function to set the value of age
        int getAge( ); // a getter function to access age data member
};
```

```
void Person::setAge(int userAge) {
        age = userAge;
}
int Person::getAge( ) {
        return age;
}

int main( ) {
        Person p1;
        p1.setAge(25);
        std::cout << p1.getAge( ) << std::endl;
}
```

Note: The double colon is a scope-resolution operator in C++ that tells the compiler that a function/method belongs to a specific class.

Exercise (5 mins)

Define an **Animal** class with the following spec:

✓ Data Members: **name** (string), **age** (int), **type** (string),
isPet (bool)

✓Methods:

1. Setter methods for all four data members
2. Getter methods for all four data members
3. A method to print a friendly intro of yourself to the animal

Test your code in main with two objects and initialize with following:

1. **(Whiskers, 3 Cat, true)**
2. **(Buddy, 5, Dog, true)**

Now imagine you have 50 data members within your class, does that mean we need to have 50 setter methods to initialize them? Absolutely not!

This is where Constructors come in.

A constructor in C++ is a special method that is automatically called when an object of a class is created.

Note: You cannot explicitly call a constructor.

A constructor has the same name as the class, it is always *public*, and it does not have any return value (not even void).

Default Constructor

```
class Course {
    private:
        std::string professorName;
        int courseNumber;
    public:
        Course( ); // default constructor
};
Course::Course( ) {
    professorName = "Mike Conti";
    courseNumber = 211;
}
```

```
int main( ) {
    Course CSC;
    Course MTH;
}
```

Parameterized Constructor

```
class Course {
    private:
        std::string professorName;
        int courseNumber;
    public:
        Course(std::string prof, int courseNum); // parameterized constructor
};
Course::Course(std::string prof, int courseNum) {
    professorName = prof;
    courseNumber = courseNum;
}
```

Another Method to define parameterized constructor (AKA Initializer lists):

```
Course::Course(std::string prof, int courseNum) : professorName(prof),
courseNumber(courseNum) { }
```

Parameterized Constructor

```
int main( ) {
    Course CSC("Michael Conti", 211);
    Course MTH("John Doe", 215);
   // etc…
}
```

The parameterized constructor is invoked/called when you create an object with specific arguments passed to initialize the data members.

Parameterized Constructor

```
class Course {
    private:
        std::string professorName;
        int courseNumber;
    public:
        Course(std::string prof, int courseNum); // parameterized constructor
};
Course::Course(std::string professorName, int courseNumber) {
    professorName = professorName;
    courseNumber = courseNumber;
}
```

Problem here is the constructor parameters and data member variables have the same name, and the compiler won't know to initialize the data members!

Parameterized Constructor

```
Course::Course(std::string professorName, int courseNumber) {
    professorName = professorName;
    courseNumber = courseNumber;
}
```

This is a common mistake called **Shadowing**—and this occurs when a parameter name in the constructor shadows the member variable name, making it difficult to access the member variable directly.

To resolve this issue, we can use the **this** pointer to refer to the member variables. **this** is a pointer that points to the object for which the member function is called.

```
Course::Course(std::string professorName, int courseNumber) {
    this->professorName = professorName;
    (*this).professorName = professorName;
}
```

Both lines are equivalent!

Copy Constructor

```
class Course {
    private:
        std::string professorName;
        int courseNumber;
    public:
         Course(std::string prof, int courseNum);
        Course(const Course &obj); // copy constructor
};
Course::Course(const Course &obj) {
    professorName = obj.professorName;
    courseNumber = obj.courseNum;
```

Copy Constructor

```
int main( ) {
    Course CSC("Michael Conti", 211);
    Course MTH = CSC;
}
```

Above, for CSC, the parameterized constructor gets called. For MTH, the copy constructor gets invoked and initializes the data members of MTH with “Michael Conti” and 211.

```
    Course CSC("Michael Conti", 211);
    Course MTH;
    MTH = CSC;
```

Note: The copy constructor does not get called here. This is an assignment operation!

INHERITANCE

Inheritance is a concept of classes that makes it possible to inherit attributes and methods from one class to another. We group the “inheritance concept” into two categories:

1. **derived class** (child): the class that inherits from another class
2. **base class** (parent): the class being inherited from

TYPES OF INHERITANCE

1. Single-level: One base class derives another class
2. Multi-level: A derived class derives another class. For example, **GrandFather** derives **Father**; **Father** derives **Child**
3. Multiple: A class is derived from more than one base class. For example, **coloredShape** is derived from **Shape** and **Color**.

```
class bigTech {
    protected:
        std::string companyName;
        int numEmployees;
    public:
        bigTech(std::string name, int numEmp) : companyName(name),
            numEmployees(numEmp) { }
};
class Netflix : public bigTech {
    private:
        int numSubscribers;
    public:
        Netflix(std::string name, int numEmp, int subs) : bigTech(name, numEmp),
            numSubscribers(subs) { }
};
```

What's the Output?

```
class Animal {
   public:
     Animal( ) {
       std::cout << "This is an animal!\n";
     }
};s
class Dog : public Animal {
   public:
     Dog( ) {
       std::cout << "This is a dog!\n";
     }
};
```

```
class Bulldog : protected Dog {
    private:
      int age;
     public:
       Bulldog(int age) {
          this->age = age;
          std::cout << "This is a bulldog!";
        }   };
int main( ) {
    Animal x;
    Animal* y = new Animal( );
    Dog d;    Bulldog b(5);   delete y;
}
```

DESTRUCTORS

Like constructors, destructors are special methods within a class. Destructors must be *public,* have the same name as the class preceded by a **~**, are automatically called when an object is destroyed, does not have a return type (not even void).

Unlike constructors, destructors takes **no** arguments.

So essentially, when an object is destroyed (meaning they exist in the stack and goes out of scope, or they exist in the heap and the **delete** keyword is used), a destructor automatically gets called.

What's the Output?

```
class Animal {
    public:
      Animal( ) {
          std::cout << "Animal Constructor\n";
      }
      ~Animal ( ) {
          std::cout << "Animal Destructor\n";
       }
};
int main( ) {
     Animal x;
}
```

What's the Output?

```
class Animal {
    public:
      Animal( ) {
          std::cout << "Animal Constructor\n";
      }
      ~Animal ( ) {
          std::cout << "Animal Destructor\n";
       }
};
int main( ) {
     Animal x;
     Dog d;
}
```

```
class Dog : public Animal {
    public:
       Dog( ) {
          std::cout << "Dog Constructor\n";
       }
       ~Dog( ) {
          std::cout << "Dog Destructor\n";
       }
 };
```