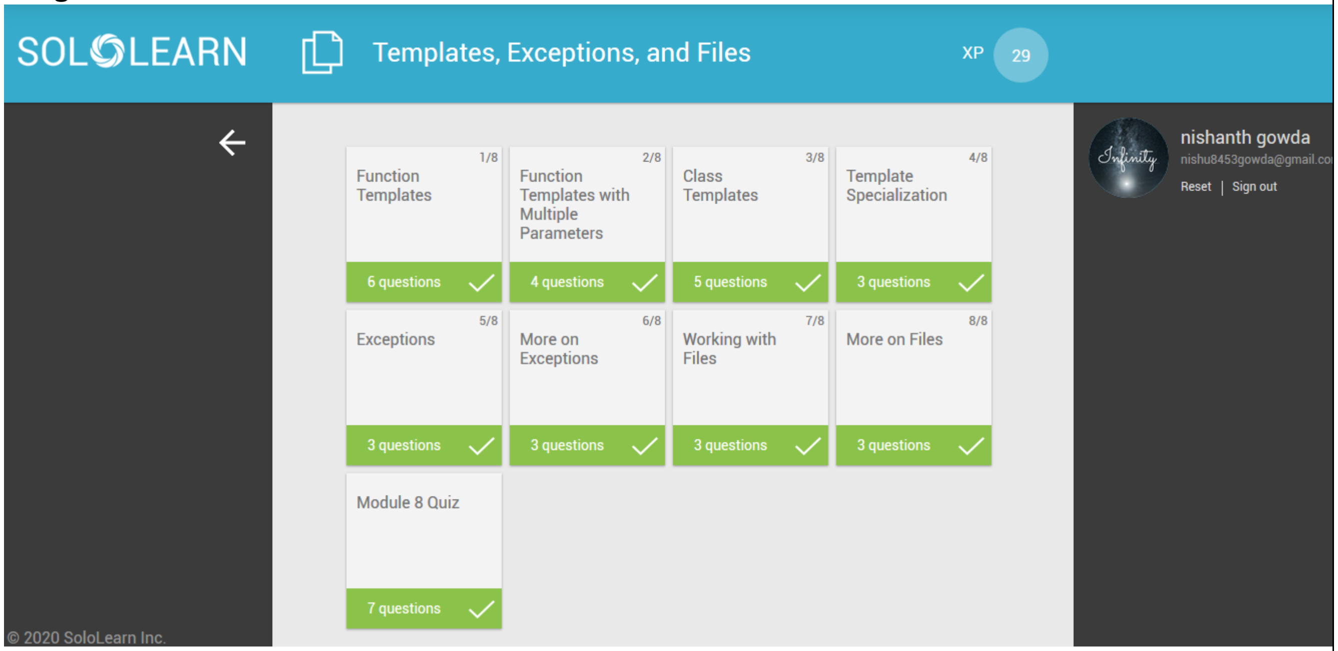
DAILY ASSESSMENT FORMAT

Date:	27/June/2020	Name:	nishanth
Course:	C++ programming	USN:	4al17ec063
Topic:	Function templates Working with files	Semester & Section:	6th b
GitHub Repository:	nishanthvr		

FORENOON SESSION DETAILS

Image of session



SOLOLEARN Templates, Exceptions, and Files XP 29

nishanth gowda
nishu8453gowda@gmail.co
Reset | Sign out

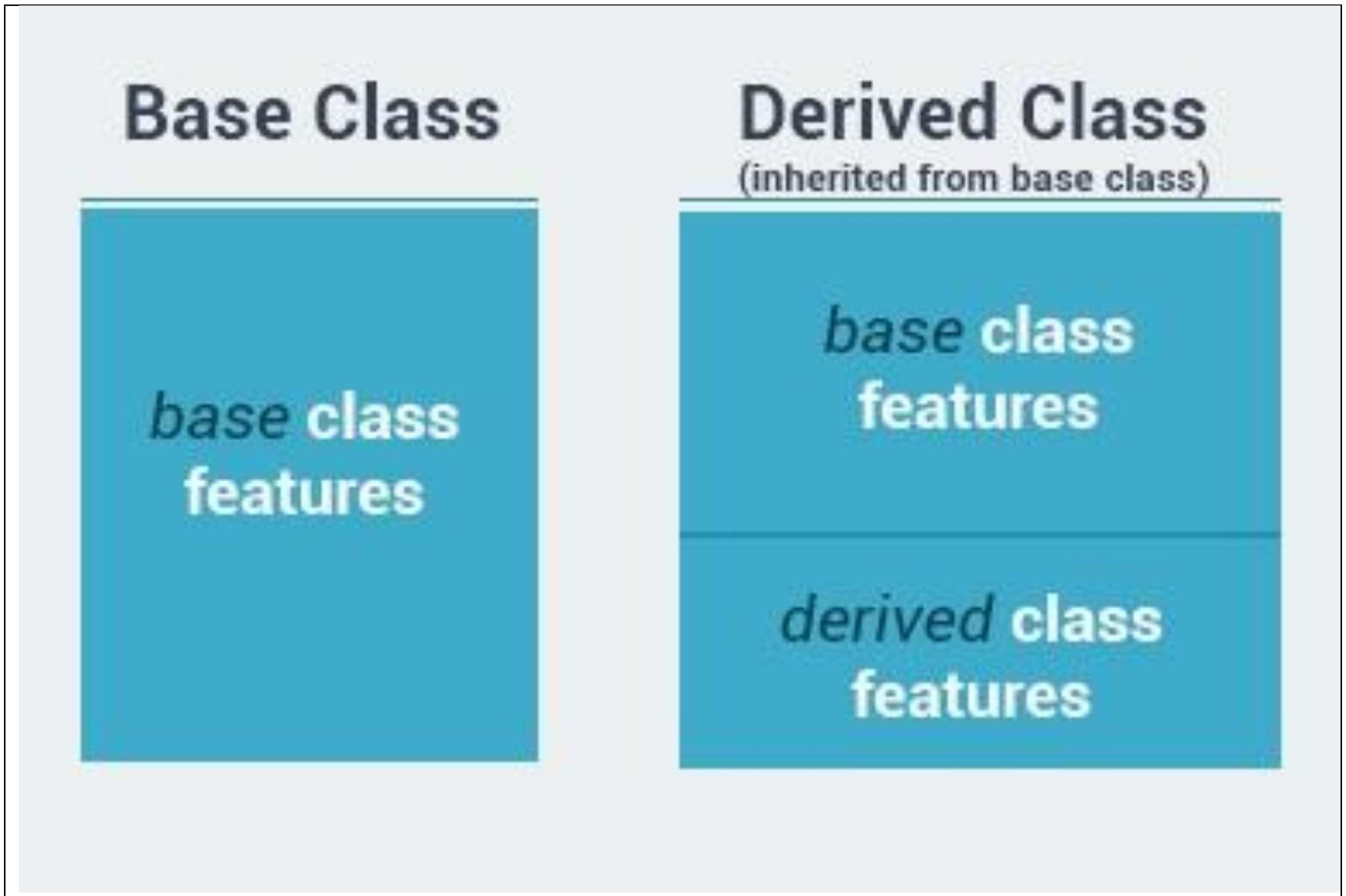
1/8 Function Templates — 6 questions ✓	2/8 Function Templates with Multiple Parameters — 4 questions ✓	3/8 Class Templates — 5 questions ✓	4/8 Template Specialization — 3 questions ✓
5/8 Exceptions — 3 questions ✓	6/8 More on Exceptions — 3 questions ✓	7/8 Working with Files — 3 questions ✓	8/8 More on Files — 3 questions ✓
Module 8 Quiz — 7 questions ✓			

© 2020 SoloLearn Inc.

Inheritance

Inheritance is one of the most important concepts of object-oriented programming.
Inheritance allows us to define a class based on another class. This facilitates greater ease in creating and maintaining an application.

The class whose properties are inherited by another class is called the **Base** class. The class which inherits the properties is called the **Derived** class. For example, the **Daughter** class (derived) can be inherited from the **Mother** class (base).
The derived class inherits all feature from the base class, and can have its own additional features.



The idea of inheritance implements the **is a** relationship. For example, mammal IS-A animal, dog IS-A mammal, hence dog IS-A animal.

Polymorphism

The word **polymorphism** means "having many forms".
Typically, polymorphism occurs when there is a hierarchy of classes and they are related by **inheritance**.

C++ polymorphism means that a call to a member function will cause a **different** implementation to be executed depending on the **type** of object that invokes the function.
Simply, polymorphism means that a single function can have a number of different implementations.
Polymorphism can be demonstrated more clearly using an example:
Suppose you want to make a simple game, which includes different enemies: monsters, ninjas, etc. All enemies have one function in common: an **attack** function. However, they each attack in a different way. In this situation, polymorphism allows for calling the same **attack** function on different objects, but resulting in different behaviors.

The first step is to create the **Enemy** class.class Enemy {
protected:
int attackPower;
public:
void setAttackPower(int a){
attackPower = a;

```
}
};
```
Our Enemy class has a public method called **setAttackPower**, which sets the protected **attackPower** member variable.
Our second step is to create classes for two different types of enemies, **Ninjas** and **Monsters**. Both of these new classes inherit from the **Enemy** class, so each has an attack power. At the same time, each has a specific **attack** function. class **Ninja**: public Enemy {
```
public:
void attack() {
cout << "Ninja! - "<<attackPower<<endl;
}
};

class Monster: public Enemy {
public:
void attack() {
cout << "Monster! - "<<attackPower<<endl;
}
};
```
As you can see, their individual **attack** functions differ.
Now we can create our **Ninja** and **Monster** objects in main. int main() {
```
Ninja n;
Monster m;
}
```
Ninja and **Monster** inherit from **Enemy**, so all **Ninja** and **Monster** objects are **Enemy** objects. This allows us to do the following:**Enemy** *e1 = &n;
```
Enemy *e2 = &m;
```
We've now created two pointers of type **Enemy**, pointing them to the **Ninja** and **Monster** objects.

Function Templates

Functions and classes help to make programs easier to write, safer, and more maintainable.
However, while functions and classes do have all of those advantages, in certain cases they can also be somewhat limited by C++'s requirement that you specify types for all of your parameters.

For example, you might want to write a function that calculates the sum of two numbers, similar to this:int sum(int a, int b) {
```
return a+b;
}
```
You can use templates to define functions as well as classes. Let's see how they work.
It becomes necessary to write a new function for each new type, such as doubles.double sum(double a, double b) {
```
return a+b;
}
```
Wouldn't it be much more efficient to be able to write one version of sum() to work with parameters of **any** type?
Function templates give us the ability to do that!
With function templates, the basic idea is to avoid the necessity of specifying an exact type for each variable.

Instead, C++ provides us with the capability of defining functions using placeholder types, called **template type parameters**.

To define a function template, use the keyword **template**, followed by the template type definition:**template** <class T>
We named our template type **T**, which is a generic data type.
Tap **Continue** to learn more!

Working with Files

Another useful C++ feature is the ability to read and write to files. That requires the standard C++ library called **fstream**.
Three new data types are defined in fstream:
ofstream: Output file stream that creates and writes information to files.
ifstream: Input file stream that reads information from files.
fstream: General file stream, with both ofstream and ifstream capabilities that allow it to create, read, and write information to files.

To perform file processing in C++, header files <**iostream**> and <**fstream**> must be included in the C++ source file.#include <iostream>
#include <fstream>
These classes are derived directly or indirectly from the classes istream and ostream. We have already used objects whose types were these classes: cin is an object of class istream and cout is an object of class ostream.