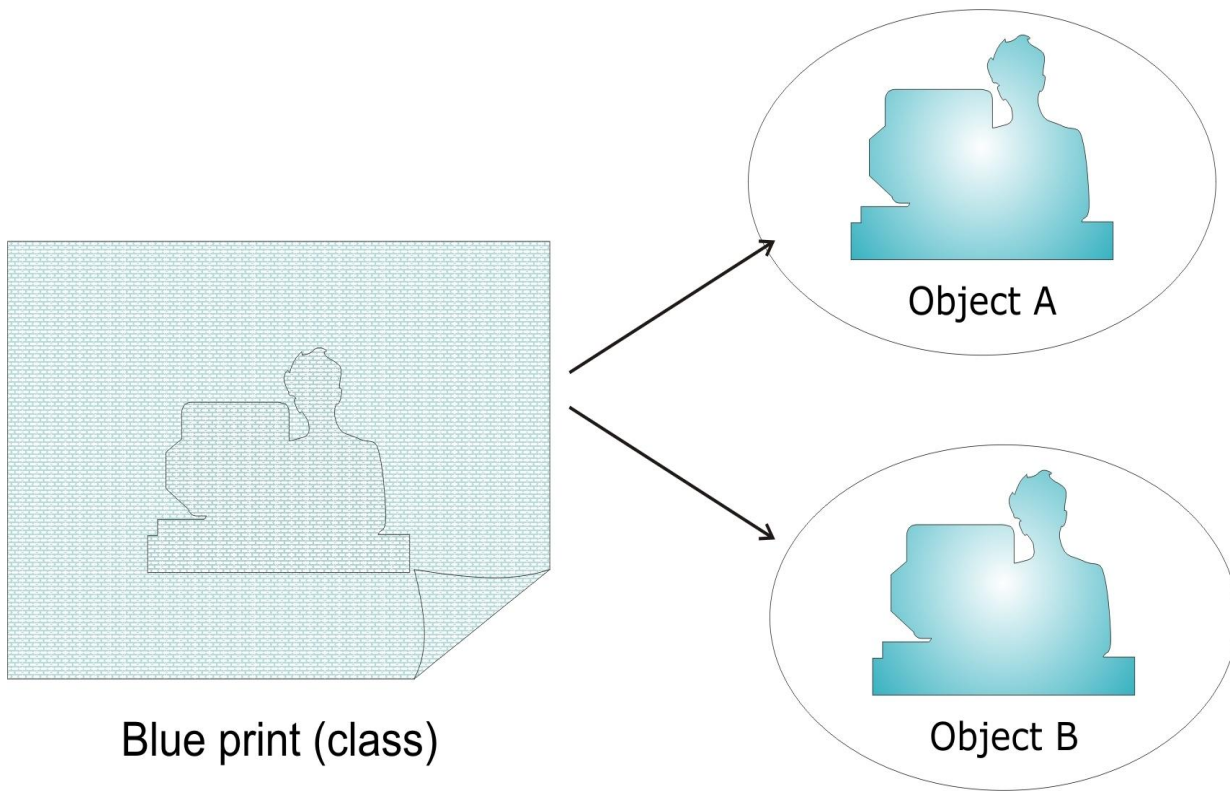


Class

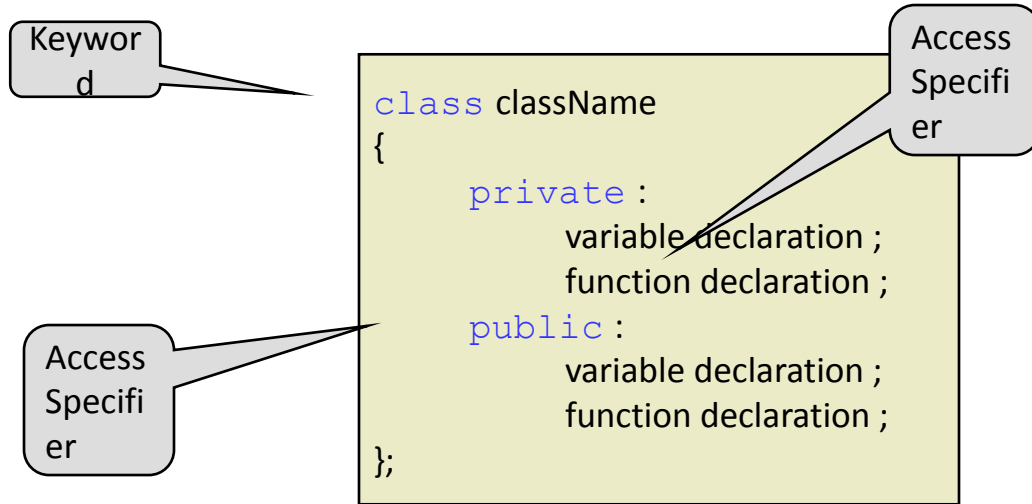
- A template for creating similar objects.
- Maps real world entities into classes through data members and member functions.
- A user defined type.
- An object is an instance of a class.
- By writing a class and creating objects of that class, one can map two concepts of object model, abstraction and encapsulation, into software domain.

Objects and Classes



Class

- Template for the creation of similar objects.
- A class in C++ is an encapsulation of data members and member functions that manipulate the data.



If semicolon is missing, compiler throws an error

- syntax error : missing ';' before 'PCH creation point' Error executing `cl.exe`

Class Components

- A class declaration consists of following components
 - Access specifiers: restrict access of class members
 - `private`
 - `protected`
 - `public`
 - Data members
 - Member functions
 - Constructors
 - Destructors
 - Ordinary member functions

Constructor

- Special member function of the class with same name as its class name.
 - Used to initialize attributes of an object
- Implicitly called when objects are created.
 - Without any input parameter is no-argument constructor.
- Rules for implementing constructor:
 - No return type for constructor. Not even void.
 - Multiple constructors can be written - different number, types and order of parameters.
 - Must have same name as that of the class.

Constructors in class cDate

```
class cDate
{
    ...
public:
//No-argument Constructor
    cDate()
    {
        mDay = 1;
        mMonth = 1;
        mYear = 2000;
    }
//Parameterized Constructor
    cDate(int d, int m, int y)
    {
        mDate = d;
        mMonth = m;
        mYear = y;
    }
}
```

```
int main()
{
    cDate d1;
    cDate d2(25, 8,
2014);
    return 0;
}
```

Calls
no-arguement
constructor

Calls parameterized
constructor.

Copy Constructor

Copy [Constructors](#) is a type of constructor which is used to create a copy of an already existing object of a class type.

It is usually of the form `X (X&)`, where `X` is the class name.

The compiler provides a default Copy Constructor to all the classes.

used to copy data of one object to another.

Implicitly-declared copy constructor

If no user-defined copy constructors are provided for a class type the compiler will always declare a copy constructor as a non-[explicit](#) inline public member of its class.

Syntax of Copy Constructor

```
Classname(const classname & objectname)
{
    . . . .
}
```

Calling copy constructor

1. A o(2,3); A obj(o);
2. A o1=o; //gives call to copy constructor

A o4; //non parameterized constructor

o4=o1 //this is not call to copy constructor // but this is call to operator overloading

Class A

{

int n;

A(int n1 = 1) {

n=n1;

}

A(const A& a) {

n=a.n;

} // user-defined copy ctor

};

int main()

{

A a1(7);

A a2(a1); // calls the copy ctor

}

Types of Member Functions

- Following are the types of member functions of a class:
 - **Mutator** - Changes contents of instance members
 - **Accessor** - Accesses instance members
 - **Facilitator** - Helps to view the values of attributes of object
 - **Helper** - A private function, accessed from public member functions of same class to help in their implementation

class cDate: Member Functions

```
class cDate
{
    ...
public:
    void display(); //Facilitator Function
    int getDay(void); //Accessor Function
    void setDay(int); //Mutator Function
    ...
};
```

```
//Function definition of display member function
void cDate :: display()
{
    cout<<"Date:"<<mDay<<"/"<<mMonth<<"/"<<
mYear<<endl;
}
```

Invoking a Member Function

```
int main()
{
    cDate d1(2,3,4);
    cDate d2(5,6,7);
    d1.display();
    d2.display();
    return 0;
}
```

this Keyword

- **this** is a keyword in C++.
- It is a pointer
- **this** always holds a reference of an object which invokes the member function.
 - **this** points to an individual object.
- It is a hidden parameter that is passed to every class member function.

Using this Keyword

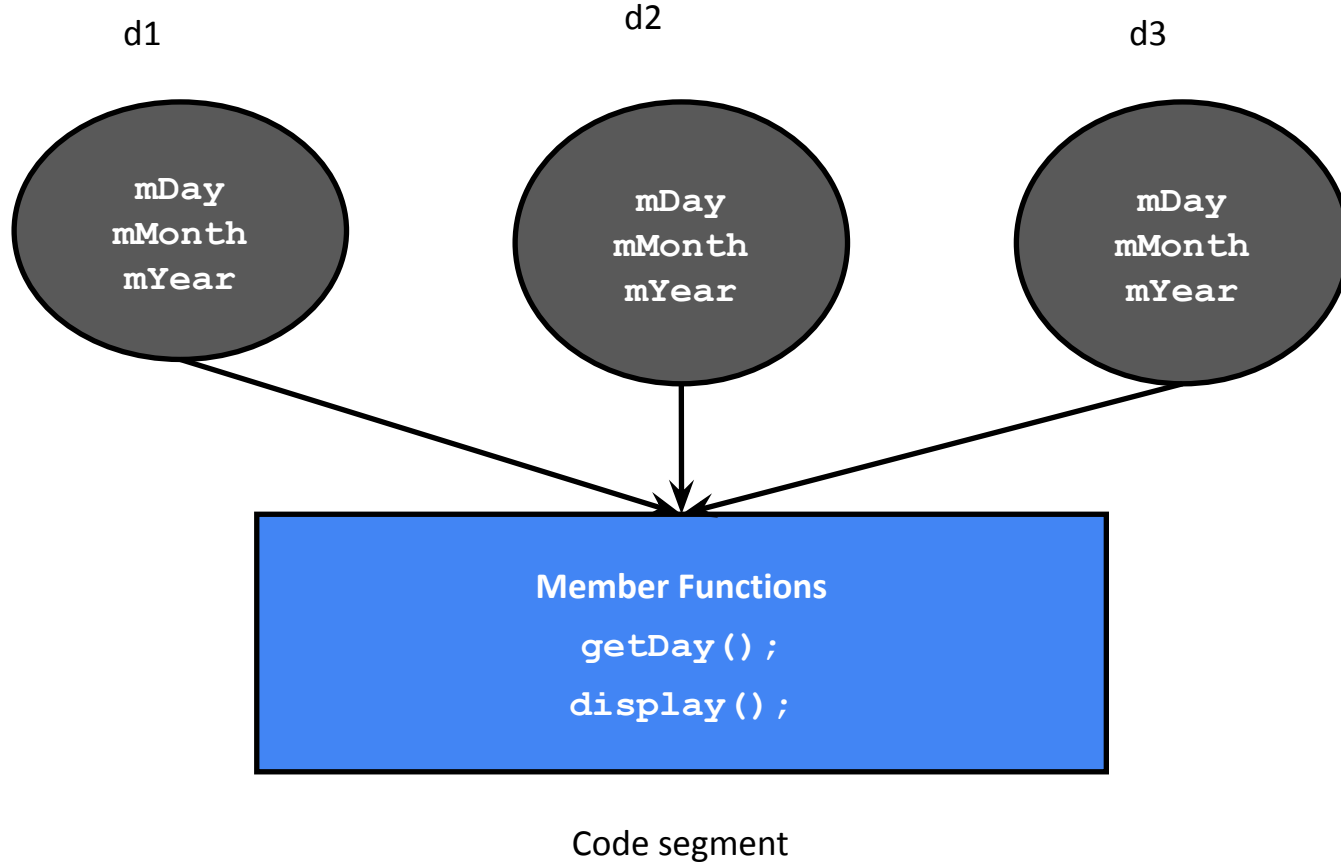
```
int cDate::getDay(void)
{
    return this->mDay;
    //OR return mDay;
}
```

```
void cDate::setDay(int d)
{
    this->mDay = d;
}
```

```
int main()
{
    cDate d1(4,5,6), d2;
    int day = d1.getDay();
    cout<< "Day is = " << day;    // displays 4
    d2.setDay(5);    // sets day of d2 to 5
    return 0;
}
```

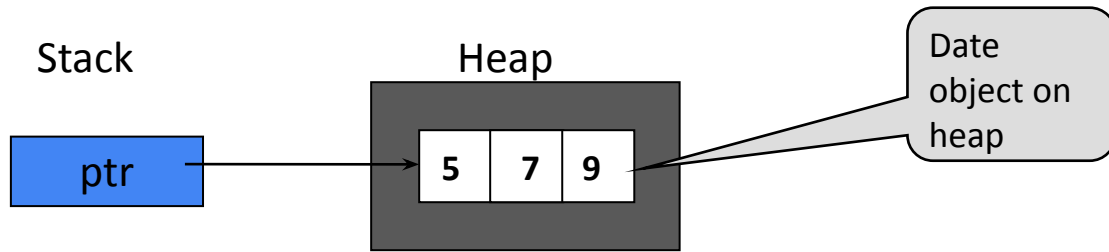
Access
current object

Memory Allocation to Objects



Creating an Object on Heap

```
int main()
{
    cDate* ptr = new cDate (5,7,9);
    ...
    delete ptr ;
    return 0;
}
```



Creating `const` Objects

- To create a constant object use `const` keyword:

```
const cDate d1(2,3,4);    // statement in main
```

- `const` objects invoke `const` member functions only.
- `const` functions are 'read only' functions.

Const member function

- Const function can be called on any type of object, const object as well as non-const objects.
- object is declared as const, it needs to be initialized at the time of declaration.
- not to allow to modify the object on which functions are called.
- Accidental changes to objects are avoided.

const Member Functions

```
//In Class Declaration
class cDate
{
    ...
public:
    ...
    int getDay(void) const;
    ...
};
```

```
//Function Definition
int cDate::getDay(void) const
{
    return this->mDay;
}
```

```
int main()
{
    const cDate d1(5,7,9);
    int day = d1.getDay();
    cout<<"Day is = "
        <<day;
    return 0;
}
```

Static Variables

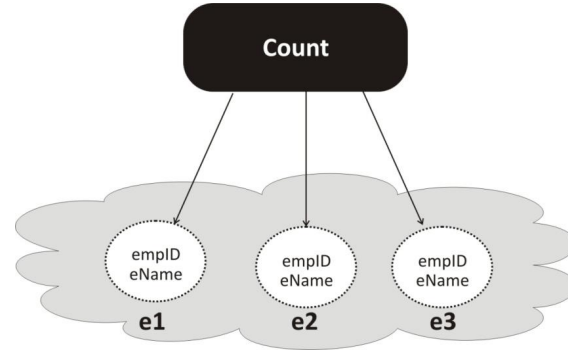
- Some characteristics or behaviors belong to the class rather than a specific instance
 - `interestRate`, `CalculateInterest` method for a `SavingsAccount` class
 - `count` variable in `Employee` to automatically generate employee id
- Such data members are static for all instances
 - Change in static variable value affects all instances
 - Also known as class variable.

Application

.To keep track how many objects created

Static Variables in Memory

```
class Employee
{
    int empId;
    int eName;
    static int count;
}
```



- Data to be shared by all objects is stored in static data members.
- Only a single copy exists.
- Class scope and lifetime is for entire program.
- How can they be accessed?

Static Member Functions

- Can access static data members only.
- Invoked using class name as:

```
class name :: functionName();
```

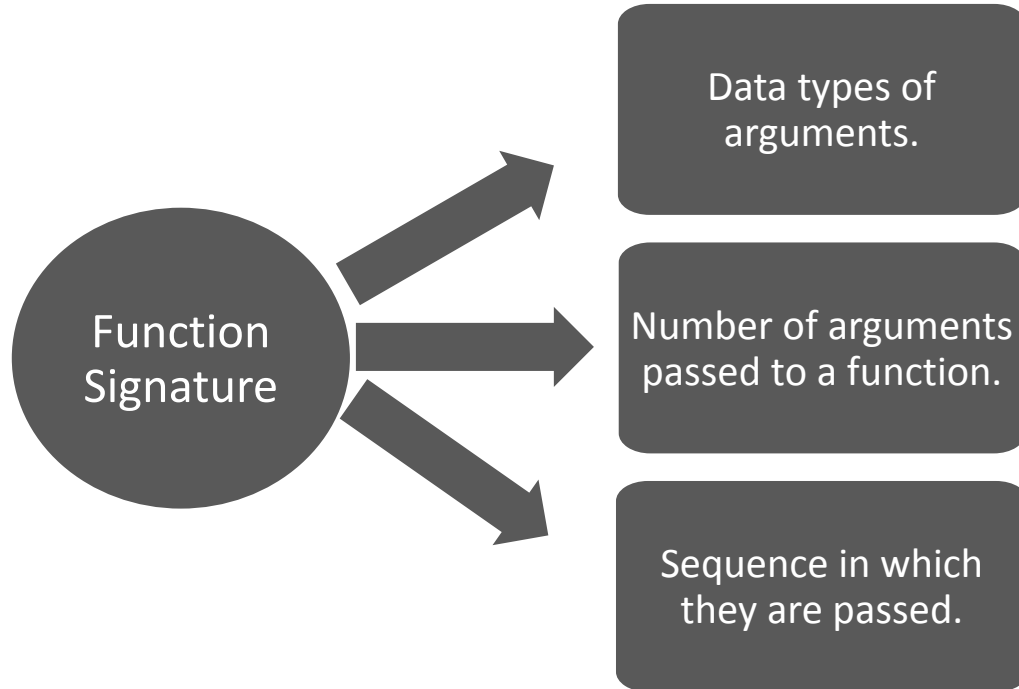
- **this** pointer is never passed to a static member function.

```
public class cEmployee
{
    . . .
    static int count;
    static int showCount()
    {
        return count;
    }
}
```

```
main()
{
    int number =
    Employee::showCount();
    cout<< "Number
    employees are:" <<
    number;
}
```

Function Overloading

- Using functions with same name but different signatures in the same program is called function overloading.



Name Mangling of Overloaded Functions

- Names of overloaded functions are mangled and may look something like this:

```
int sum(int a, int b)      sum@1.....  
float sum (float a, float b)  sum@2.....  
float sum (int a, float b)    sum@3.....  
float sum (float a, int b)    sum@4.....  
Void sum(int a, int b, int c)  sum@5.....  
int sum(int a, int b, int c)  //Not Fun. Overloading
```


Name mangling Example

```
// Name Mangling in function overloading
```

```
int f(void) { return 1; }
```

```
int f(int) { return 0; }
```

```
void g(void) { int i = f(), j = f(0); }
```

Name mangling

```
int __f_v(void) { return 1; }
```

```
int __f_i(int) { return 0; }
```

```
Void g_v(void) { int i=__f_v(), j =  
__f_i(0); }
```

Function Overloading

- While using function overloading note that:
 - Each function in C++ is name mangled.
 - Name mangling algorithm is different for different compilers, e.g. Microsoft , Borland.
 - Therefore, C++ code compiled under different compilers may not be compatible.
 - Use `extern "C"` directive to suppress name mangling.

Operator Overloading

The mechanism of giving special meaning to an operator is known as operator overloading.

For example, we can overload an operator '+' in a class like string to concatenate two strings by just using +.

Implementation of Operator overloading:

1. Member function: It is in the scope of the class in which it is declared.
2. Friend function: It is a non-member function of a class with permission to access both private and protected members.

Rule

- To work, at least one of the operand must be a user-defined class object.
- We can only overload the existing operators, Can't overload new operators.
- Some operators cannot be overloaded using a friend function. However, such operators can be overloaded using the member function.

Which operators Cannot be overloaded?

- Conditional [?:], size of, scope(::), Member selector(.), member pointer selector(.*) and the casting operators.
- We can only overload the operators that exist and cannot create new operators or rename existing operators.

- At least one of the operands in overloaded operators must be user-defined, which means we cannot overload the minus operator to work with one integer and one double. However, you could overload the minus operator to work with an integer and a mystring.
- It is not possible to change the number of operands of an operator supports.
- All operators keep their default precedence and associations (what they use for), which cannot be changed.
- Only built-in operators can be overloaded.

Syntax

RT operator Symbol(DT)

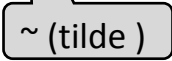
{

}

Destructor

- Destructor is a special member function of the class that is invoked implicitly to release the resources held by the object.

```
~cComplex( ); or ~cString( );
```

- Characteristics:  `~ (tilde)`
 - Has same name as that of class.
 - Does not have a return type or parameters.
 - Cannot be overloaded. Therefore a class can have only one destructor.
 - Implicitly called whenever an object ceases to exist.

Destructor

- Destructor function de-initializes the objects when they are destroyed.
- It is automatically invoked
 - when object goes out of scope or
 - when the memory allocated to object is de-allocated using the `delete` operator.
- It is used to release the resources occupied by the object.
 - If a class contains pointer as a data member then it is mandatory on programmers part to implement a destructor otherwise there is problem of memory leakage.

Containment:

- We can create an object of one class into another and that object will be a member of the class.
- This type of relationship between classes is known as **containership** or **has_a** relationship as one class contain the object of another class.
- Containment represents 'has a' or 'is a part of' relationship.
- Containment relationship depicts how an object may be a part of another object.
 - For example,
Engine, wheels, etc. are a part of a car.
Pan card is required to open a bank account
- Container relationship brings reusability of code.
 - For example,
Engine is also used in an airplane.
Pan card is also used in I-T returns.

Class cEmployee

```
// container class
class Employee
{
protected: // accessible in derived classes
    int mId;
    int mBasicSal;
    Date mBdate; // contained object
public :
    Employee(){ // no-argument c'tor
        mId = 0;
        mBasicSal = 0;
    }
    cEmployee(int, int, int, int, int);
    void display();
};
```

```
class cDate {
private:

    int mDay, mMon, mYear;
public:
    cDate();
    cDate(int, int, int);
    void display();
    . . .
    void setDay(int);
};
```

Constructor for the Class `cEmployee`

```
Employee::Employee(int i, int sal,  
int d, int m, int y){  
    mId = i;  
    mBasicSal = sal;  
  
    mBdate = Date(d, m, y);  
}
```

Uses constructor
conversion function
to convert built-in
type to class-type

```
int main()  
{  
    Employee e1( 1, 10000,17,01,1970);  
    return 0;  
}
```

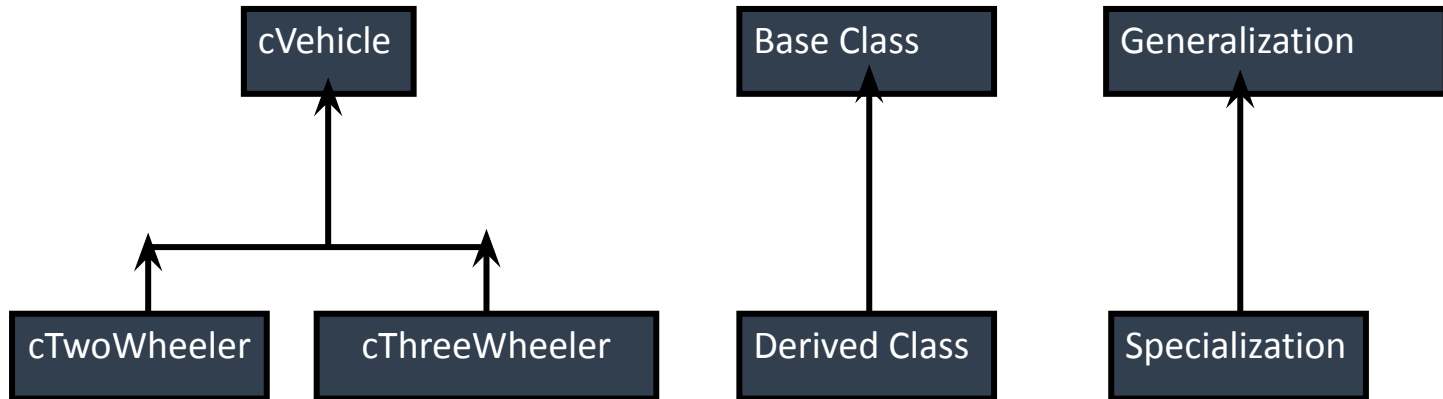
Calls
parameterized
constructor

Inheritance

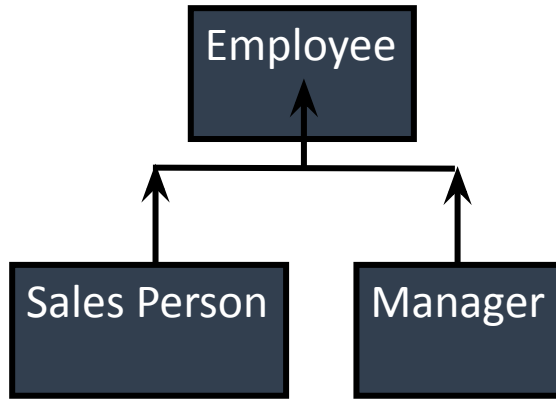
- One of the key-concepts of object-oriented approach.
- Allows creation of hierarchical classification.
- In C++, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which are defined in other class.
- In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.
- Advantages of Inheritance
 - Reusability
 - Extensibility

Base and Derived Class

Derived class inherits all data members and methods of its base class.



Base and Derived Classes - Example



- This is 'is a' kind of hierarchy.
- More than one class can inherit attributes from a single base class.
- A derived class can be a base class to another class.

- **C++ supports five types of inheritance:**

- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance
- Hybrid inheritance

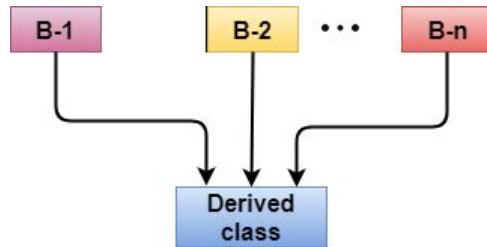
Single inheritance is defined as the inheritance in which a derived class is inherited from the only one base class.

Class B---> Class A

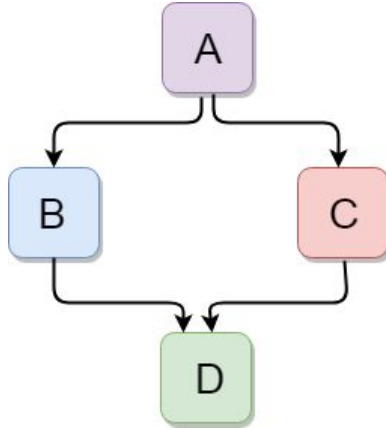
Multilevel inheritance is a process of deriving a class from another derived class.

Class C—>Class B—>Class A

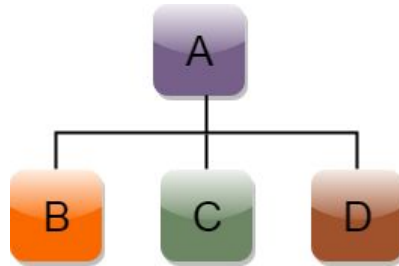
Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.



- Hybrid inheritance is a combination of more than one type of inheritance.



- Hierarchical inheritance is defined as the process of deriving more than one class from a base class.



Inheritance Syntax

- Access can be **public** or **private** or **protected**.

```
class baseClassName
{
    // body of base class
};
class derivedClassName:access baseClassName
{
    // body of derived class
};
```

Modes of Inheritance

- Private

- Access specifier: `private`
- Private members of base class are not accessible in derived class
- Protected members of base class treated as private in derived class.
- All public members of base class are treated as private in derived class

- Protected

- Access specifier: `protected`
- Private members of base class are not accessible in derived class
- Protected members of base class treated as protected in derived class.
- All public members of base class are treated as protected in derived class

Inheritance Mode :Public

- Class Derived: `public Base_Class`
- All public members of base class become public members of derived class
- All protected members of base class become protected members of derived class
- All private members of base class remain private to base class.

Base class member access specifier	Type of Inheritance		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Derived Class Constructors/Destructors

- Constructors are called in the sequence of
 - Base -> Derived
- Destructors are called in the sequence of
 - Derived -> Base

Constructors in Derived Classes

- If the base class constructor does not have any arguments, there is no need for any constructor in the derived class
- if there are one or more arguments in the base class constructor, derived class need to pass argument to the base class constructor
- If both base and derived classes have constructors, base class constructor is executed first
- In multiple inheritances, base classes are constructed in the order in which they appear in the class deceleration
- In multilevel inheritance, the constructors are executed in the order of inheritance
- C++ supports a special syntax for passing arguments to multiple base classes

- The constructor of the derived class receives all the arguments at once and then will pass the call to the respective base classes
- The body is called after the constructors is finished executing

syntax:

```
Derived-Constructor (arg1, arg2, arg3....): Base 1-Constructor (arg1,arg2),  
Base 2-Constructor(arg3,arg4)
```

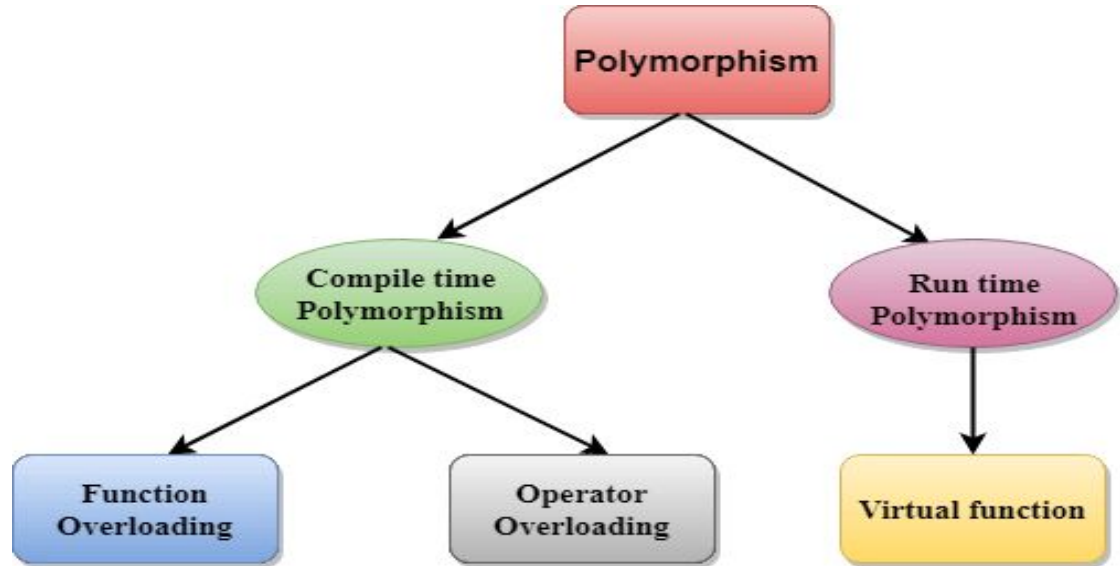
```
{
```

```
....
```

```
}
```

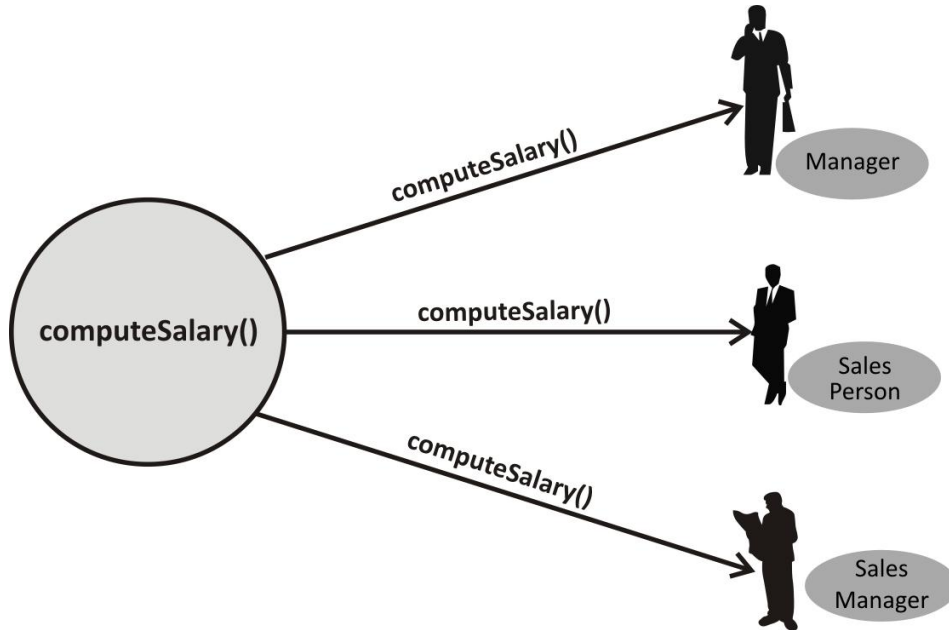

Polymorphism

The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms. It is a greek word. In object-oriented programming, we use 3 main concepts: inheritance, encapsulation, and polymorphism.



Polymorphism (Late Binding)

- Ability of different related objects to respond to the same message in different ways is called polymorphism.



Compile-time Binding and Run-time Binding

- Binding is an association of function call to an object.
- Compile-time binding
 - The binding of a member function call with an object at compile-time.
 - Also called static type or early binding.
- Run-time binding
 - The binding of the function call to an object at run time.
 - Also called dynamic binding or late binding.
 - Achieved using virtual functions and inheritance.

Virtual Function

- To implement late binding, the function is declared with the keyword `virtual` in the base class.
- Points to note:
 - Virtual function is a member function of a class.
 - Virtual functions can be redefined in the derived class as per the design of the class.
 - Also considered virtual by the compiler

Virtual Function

- It is used to tell the compiler to perform dynamic linkage or late binding on the function.
- There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.
- A 'virtual' is a keyword preceding the normal declaration of a function.
- When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

Generic Pointers

Base Class pointer can point at a derived object

```
int main ()
{

    cEmployee e1(...), *pemp;
    cTrainer sp1(...);

    pemp = &e1;
    pemp = &sp1;

    pemp->calSalary();
    return 0;
}
```

pemp is a
pointer to
base class

Base class or
generic pointer can
point to objects of
derived class.

Method Overloading Vs Overriding

	Overloading	Overriding
Scope	In the same class	In the inherited classes
Purpose	Handy for program design as different method names need not be remembered	Message is same but its implementation needs to be specific to the derived class
Signature of methods	Different for each method overloaded	Has to be same in derived class as in base class
Return Type	Can be same or different as it is not considered	Return type also needs to be same

Virtual Functions

- Some points to note:
 - Should be non-static member function of the base class
 - Generally functions that are overridden in the derived class are declared as virtual functions in the base class.
 - Constructors cannot be declared as `virtual`
 - If a function is declared as `virtual` in the base class then, it will be treated as virtual in the derived class even if the keyword `virtual` is not used.

Pure Virtual Function

- A virtual function without any executable code
- Declared by using a pure specifier (`= 0`) in the declaration of a virtual member function in the class declaration.
- For example, in class `cEmployee`

```
virtual float computeSalary() = 0;
```

- A class containing at least one pure virtual function is termed as abstract class.

Types of Classes

- Concrete class

- A class which describes the functionality of the objects

- Abstract class

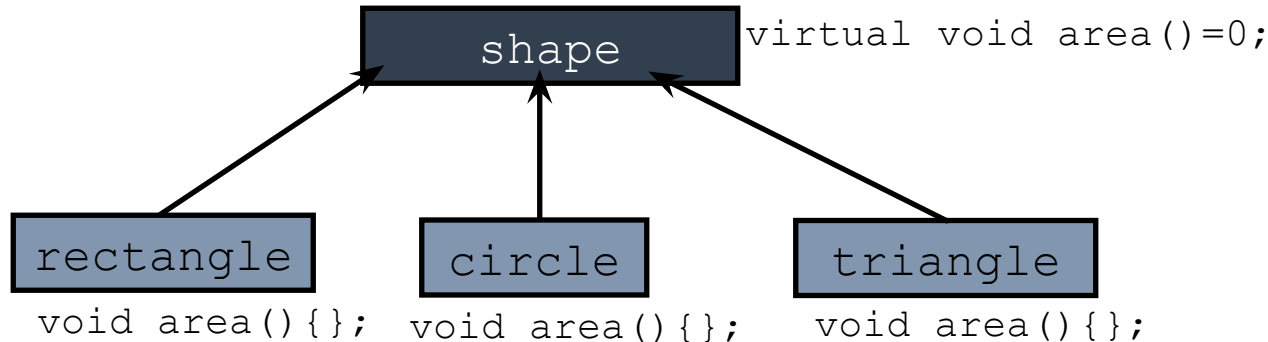
- A class which contains generic or common features that multiple derived classes can share.
- Cannot be instantiated

- Pure abstract class

- All the member functions of a class are pure virtual functions.
- It is just an interface and cannot be instantiated.

Abstract Class

- An object of an abstract class cannot be created.
 - However, pointer or reference to abstract class can be created.
 - Therefore, abstract classes support run-time polymorphism.
- Pure virtual functions must be overridden in derived classes; otherwise derived classes are treated as also abstract.



Modes of Inheritance

- Private

- Access specifier: `private`

- Public

- Access specifier: `public`

- Protected

- Access specifier: `protected`

Points to Remember . . .

- Only public inheritance is 'is a' kind of relationship.
- Private inheritance is the last stage of inheritance.
- It is not true inheritance.
 - Only advantage of private inheritance is reusability of code.
 - Do not use private inheritance. Use containment in that case.
 - Private inheritance is not 'is a' kind of inheritance.

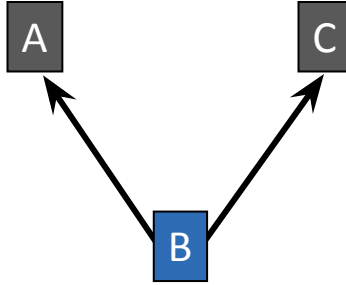
Types of Inheritance



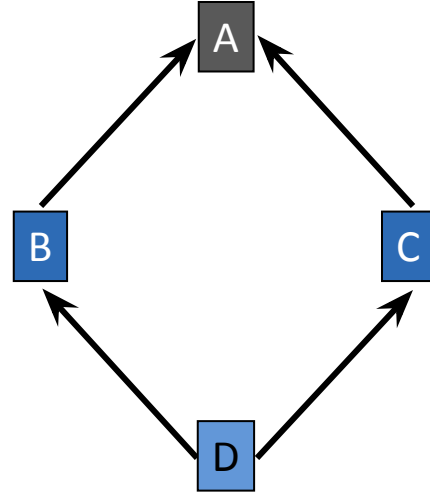
Single
Inheritance



Multi-level
Inheritance



Multiple
Inheritance

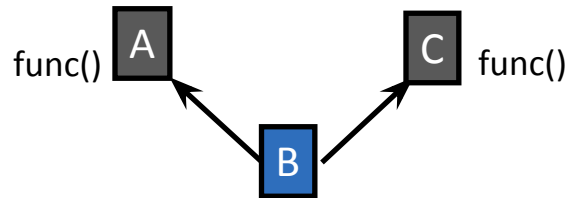


Diamond
Inheritance

Problems of Multiple Inheritance

```
class B : public A, public C { ... }
```

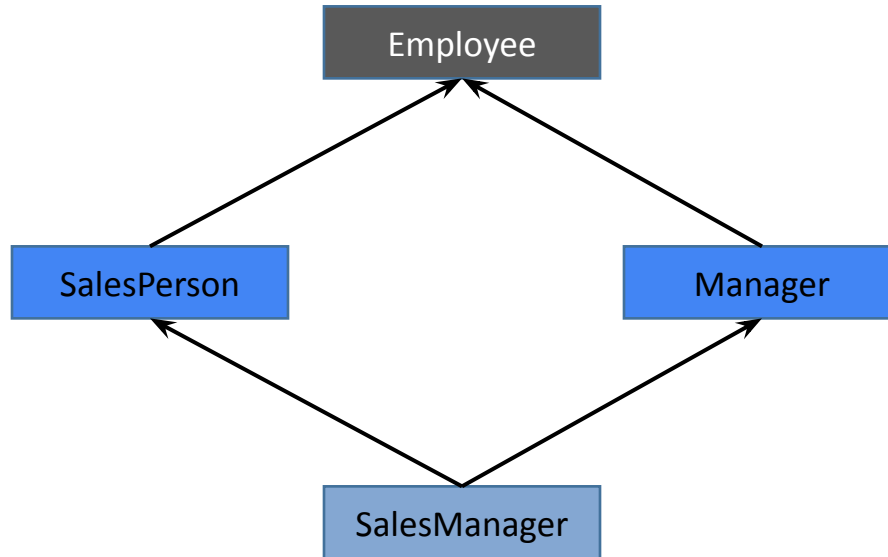
```
B bobj;  
bobj.func();
```



1. If multiple base classes contain a function with same name.
 - Resolve by any of the following ways:
 - Use scope resolution operator - `bobj.A::func()` or `bobj.C::func()`
 - Override `func()` in B class.
2. Leads to serious problem of diamond inheritance.
 - Resolve using virtual base class.

Diamond Inheritance: Derive Sales Manager

- When a class inherits from two classes, each of which inherits from a single base class, it leads to a diamond shaped inheritance pattern.



Problem: Ambiguities in Diamond Inheritance

- Case 1: what happens when two base classes contain a function with same name?

For example, `cSalesPerson` as well as `cManager` class contain `getName()` function.

```
int main()
{
    cSalesManager sm1;
    sm1.getName();

    sm1.cManager :: getName();
    return 0;
}
```

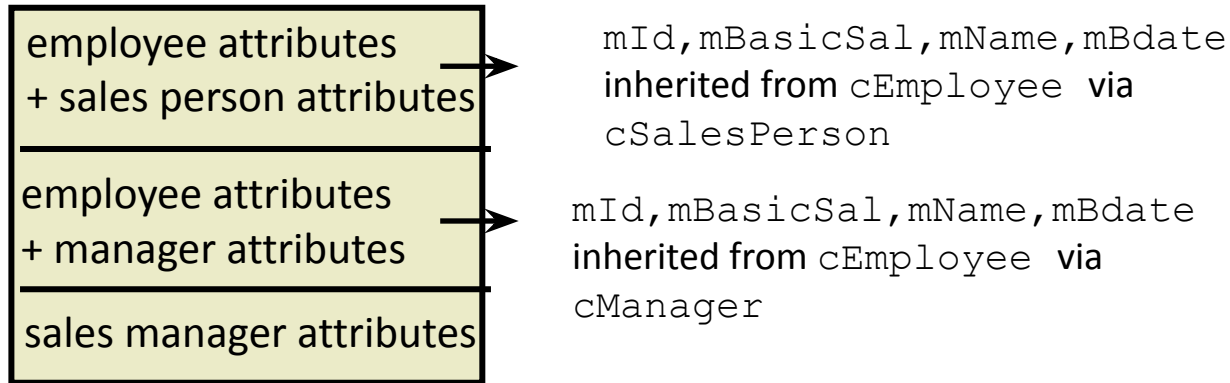
Error: Ambiguous call. Whose `getName()` will be invoked?

Resolve the ambiguity using scope resolution operator

Problem: Ambiguities in Diamond Inheritance

- Case 2 : when derived class has multiple copies of the same base class.

For example, `cSalesManager` has multiple copies of data members of `cEmployee` class.



Solution: Virtual Base Class

- Duplicate data member ambiguity can be resolved by declaring a virtual base class.
 - Derive `cSalesPerson` and `cManager` using `virtual` keyword and then derive `cSalesManager` from these two.
- By declaring base class as `virtual`, multiple copies of base class data member are not created.
- There is only one copy of common base class data members in memory and its pointer reference is there in the derived class object.

Virtual Inheritance

- The meaning of `virtual` keyword is overloaded.
- The `virtual` keyword appears in the base lists of the derived classes.

```
class cManager: virtual public cEmployee{  
    .....  
};  
class cSalesPerson: virtual public cEmployee {  
    .....  
};
```

- The most-derived constructor is responsible for initializing the `virtual` base class.

```
cSalesManager::cSalesManager(...) : cEmployee(...),  
cSalesPerson(...), cManager(...) { ... }
```

Templates

- Blueprint or formula for creating a generic class or a function.
- can create a single function or single class to work with different data types using templates.
- it gets expanded at compilation time, just like macros and allows a function or class to work on different data types without being rewritten.
- Generic programming is a technique where generic types are used as parameters in algorithms so that they can work for a variety of data types.

- Templates are powerful features of C++ which allows us to write generic programs.
- There are two ways we can implement templates:
 - Function Templates
 - Class Templates
- Similar to function templates,
- class templates used to create a single class to work with different data types.
- class templates come in handy as they can make our code shorter and more manageable.

Function Templates

- A function template starts with the keyword `template` followed by template parameter(s) inside `<>` which is followed by the function definition.

```
template <class T>
```

```
T functionName(T parameter1, T parameter2, ...) {
```

```
    // code
```

```
}
```

- In the above code, `T` is a template argument that accepts different data types (`int`, `float`, etc.), and `class` is a keyword.
- When an argument of a data type is passed to `functionName()`, the compiler generates a new version of `functionName()` for the given data type.

Calling function

```
functionName<dataType>(parameter1, parameter2,...);
```

For example, let us consider a template that adds two numbers:

```
template <class T>  
T add(T num1, T num2) {  
    return (num1 + num2);  
}
```

We can then call it in the main() function to add int and double numbers.


```
int main() {  
    int result1;  
    double result2;  
    // calling with int parameters  
    result1 = add<int>(2, 3);  
    cout << result1 << endl;  
  
    // calling with double parameters  
    result2 = add<double>(2.2, 3.3);  
    cout << result2 << endl;  
    return 0;  
}
```

```
#include<iostream>
```

```
template<typename T>
```

```
T add(T num1, T num2) {
```

```
    return (num1 + num2);
```

```
}
```

```
int main() {
```

```
    ... ..
```

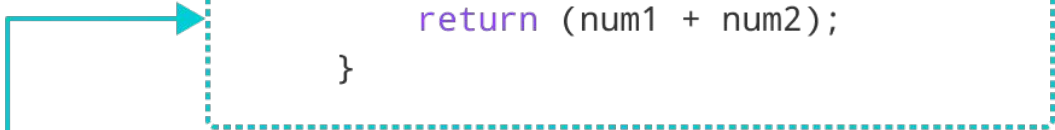
```
    result1 = add<int>(2,3);
```

```
    ... ..
```

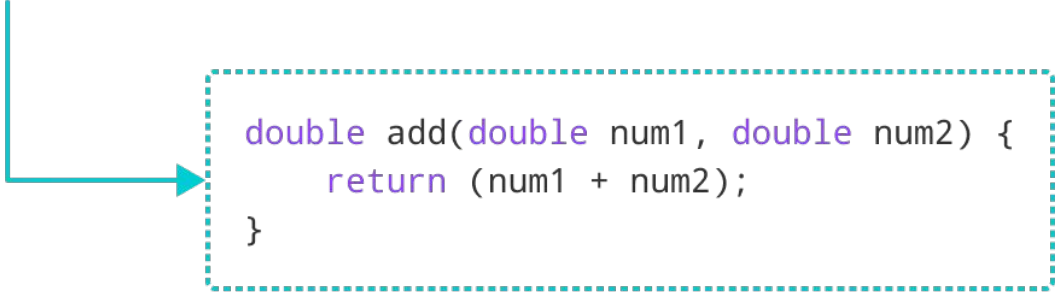
```
    result2 = add<double>(2.2,3.3);
```

```
    ... ..
```

```
}
```



```
int add(int num1, int num2) {  
    return (num1 + num2);  
}
```



```
double add(double num1, double num2) {  
    return (num1 + num2);  
}
```

Class Template

A class template starts with the keyword `template` followed by template parameter(s) inside `<>` which is followed by the class declaration.

```
template <class T>
class className {
    private:
        T var;

        ... ..

    public:
        T functionName(T arg);

        ... ..

};
```

In the above declaration, `T` is the template argument which is a placeholder for the data type used, and `class` is a keyword.

Inside the class body, a member variable `var` and a member function `functionName()` are both of type `T`.

Creating a Class Template Object

syntax

```
className<dataType> classObject;
```

For example,

```
className<int> classObject;
```

```
className<float> classObject;
```

```
className<string> classObject;
```

Defining a Class Member Outside the Class Template

```
template <class T>
class ClassName {
    ... ..
    // Function prototype
    returnType functionName();
};

// Function definition
template <class T>
returnType ClassName<T>::functionName() {
    // code
}
```

C++ Class Templates With Multiple Parameters

In C++, we can use multiple template parameters and even use default arguments for those parameters. For example,

```
template <class T, class U, class V = int>
```

```
class ClassName {
```

```
    private:
```

```
        T member1;
```

```
        U member2;
```

```
        V member3;
```

```
        ... ..
```

```
    public:
```

```
        ... .. }
```