

C++ Beyond C

Team Emertxe





C++

Agenda




```
int main(void) {
```

- Introduction to C++
- C++ vs C
- Concepts of Object Oriented Programming (OOP)
- Pillars of OOP
 - ✓ Abstraction
 - ✓ Encapsulation
 - ✓ Inheritance
 - ✓ Polymorphism
- C++ features
- Virtual Functions and Classes
- Friend Functions
- Templates - Function and Class
- STL (Standard Template Libraries)

```
}
```

Introduction to C++






C++

Introduction



- The C++ was invented by Bjarne Stroustrup in 1979 at Bell Laboratories
- Bjarne Stroustrup initially called the new language as "C with Classes". However, in 1983 the name was changed to C++.
- C++ is a statically typed, compiled, general purpose, case-sensitive programming language that supports procedural, object-oriented, and generic programming.
- Adds many new features to the C language, and is perhaps best thought of as a super-set of C.



C++

Introduction - Applications



- If you need high performance and precise control over memory and other H/W resources, then C++ would be nice choice
- Few applications of C++ are
 - Video games
 - High-performance Financial applications
 - Graphical Applications and Simulations
 - Productivity / Office Tools
 - Embedded and Real Time Software
 - Audio and Video Processing

C++ vs C

C++

- Vs C

- C++ is mainly Object Oriented Programming language, while C is Procedure Oriented Programming language.

POP	OOP
<ul style="list-style-type: none">• POP follows a top-down approach.	OOP takes a bottom-up approach in designing a program.
<ul style="list-style-type: none">• Program is divided into small chunks based on the functions.	Program is divided into objects depending on the problem.
<ul style="list-style-type: none">• Each function contains different data.	Each object controls its own data.
<ul style="list-style-type: none">• Follows a systematic approach to solve the problem.	Focuses on security of the data irrespective of the algorithm.
<ul style="list-style-type: none">• No easy way for data hiding.	Data hiding is possible in OOP

C++

- Vs C



C	C++
<ul style="list-style-type: none">• When compared to C++, C is a subset of C++.	<ul style="list-style-type: none">• C++ is a superset of C. C++ can run most of C code while C cannot run C++ code.
<ul style="list-style-type: none">• C supports procedural programming paradigm for code development.	<ul style="list-style-type: none">• C++ supports both procedural and object oriented programming paradigms; therefore C++ is also called a hybrid language.
<ul style="list-style-type: none">• In C, data are free entities and can be manipulated by outside code. This is because C does not support information hiding.	<ul style="list-style-type: none">• In C++, Encapsulation hides the data to ensure that data structures and operators are used as intended.
<ul style="list-style-type: none">• C does not allow functions to be defined inside structures.	<ul style="list-style-type: none">• In C++, functions can be used inside a structure.
<ul style="list-style-type: none">• C does not have namespace feature.	<ul style="list-style-type: none">• C++ uses NAMESPACE which avoid name collisions.

C++

- Vs C

C	C++
<ul style="list-style-type: none">• C uses functions for input/output. For example scanf and printf.	<ul style="list-style-type: none">• C++ uses objects for input output. For example cin and cout.
<ul style="list-style-type: none">• C does not support reference variables.	<ul style="list-style-type: none">• C++ supports reference variables.
<ul style="list-style-type: none">• C provides malloc() and calloc() functions for dynamic memory allocation, and free() for memory de-allocation.	<ul style="list-style-type: none">• C++ provides new operator for memory allocation and delete operator for memory de-allocation.
<ul style="list-style-type: none">• C does not provide direct support for error handling (also called exception handling)	<ul style="list-style-type: none">• C++ provides support for exception handling. Exceptions are used for "hard" errors that make the code incorrect.
<ul style="list-style-type: none">• C has no support for virtual and friend functions.	<ul style="list-style-type: none">• C++ supports virtual and friend functions.

C++

Anatomy of a Simple C++ Code

```
/* My first C++ code */
```

File Header

```
#include <iostream>
```

Preprocessor Directive

```
int main()
```

The start of program

```
{
```

```
    // To display Hello world
```

Comment


```
    std::cout << "Hello world\n";
```

Statement

```
    return 0;
```

Program Termination

```
}
```




C++

Introduction - Compilation



- Assuming your code is ready, use the following commands to compile the code
- On command prompt, type
- `$ g++ <file_name>.cpp`
- This will generate a executable named `a.out`
- But it is recommended that you follow proper conversion even while generating your code, so you could use
- `$ g++ <file_name>.cpp -o <file_name>`
- This will generate a executable named `<file_name>`



C++

Introduction - Execution



- To execute your code you shall try
- `$./a.out`
- If you have named your output file as your `<file_name>` then
- `$./<file_name>`
- This should be the expected result on your system

Object Oriented Programming System(OOPs)





- An Object is a real-world entity which will have state and behavior.
- Object is an instance of a Class.
- Object also comes with its own identity.
- So if we consider a Car as an example, then
 - Attributes : Color, Type
 - Behavior : Starts, Moves, Stops ...
 - Identity : Ford, Audi
- One cars behavior/attribute will not affect other cars. So every car has its own identity.

- Now in computing world, the objects are not physical or even visible (some times) that's it!, but every other aspect remains the same as real time object
- Some example can be like
 - Date & Time: Can be seen and it changes
 - Timer: A peripheral register who's value changes, using which we derive date and time, but generally not seen by the end user
- Well how do I create objects in C++?

- The building block of C++ that leads to Object-Oriented programming is a Class.
- It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.
- A class is like a blueprint for an object.
- For Example:
 - Consider the Class of Cars. There may be many cars with different names and brand but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range etc.
- A technique which helps to describe the object completely from properties to its implementation.

C++

OOPs Concepts - Class



- What is to be understood here is the blueprint of bicycle will always be same, like its going to have 2 tires, a seat, a handle etc.,
- We may create different types of bicycles with a defined class.
- Class Components:
 - Identity - what is it known as?
 - Employee, Bank Account, Event, Player, Document, Album
 - Attributes - what describes it?
 - Width, Height, Color, Score, File Type, Length
 - Behavior - what can it do?
 - Play, Open, Search, Save, Print, Create, Delete, Close



C++

OOP Concepts - Class - Components



Balance
Number
Holder name
Date opened

Deposit()
Withdraw()
Transfer()

Class

10000
SBIN001122
Ajay
03/05/2000

Deposit()
Withdraw()
Transfer()

Object

150
SBIN123456
John
25/11/2015

Deposit()
Withdraw()
Transfer()

Object



C++

Class

Syntax

```
class ClassName
{
    /* Group of data types */
    /* Group of functions */
};
```


Syntax

```
class Employee
{
    int id;
    string name;
    string address;

    void get_id(void)
    {
        cout << "Enter ID No: ";
        cin >> id;
    }
    void get_name(void)
    {
        cout << "Enter Name: ";
        cin >> name;
    }
    void get_address(void)
    {
        cout << "Enter Address: ";
        cin >> address;
    }
};
```

Access Specifiers/Modifiers





C++

Class - Access Modifiers



- Access modifiers are used to set the boundaries or accessibility of the class members.
- 3 Types:
 - Private
 - ◆ This is the default access behavior
 - ◆ Accessed only by the functions inside the class
 - ◆ Not allowed to be accessed directly by any object or function outside the class
 - Public
 - ◆ Can be accessed by all members, even by other classes too
 - Protected
 - ◆ Similar to that of private access modifiers, the difference is that the class member declared as Protected are inaccessible outside the class but they can be accessed by any subclass(derived class) of that class

C++

Class - Example

001_example.cpp

```
#include <iostream>

using namespace std;

class Employee
{
private:
    // Members
    int id;
    string name, address;

public:
    // Methods
    void get_data(void)
    {
        cout << "Enter ID No: "; cin >> id;
        cout << "Enter Name: "; cin >> name;
        cout << "Enter Address: "; cin >> address;
    }
    void print_data(void)
    {
        cout << "The ID is: " << id << endl;
        cout << "The Name is: " << name << endl;
        cout << "The Address is: " << address << endl;
    }
};
```

```
int main()
{
    Employee emp1;

    emp1.get_data();
    emp1.print_data();

    return 0;
}
```

C++

Class - Example

002_example.cpp

```
#include <iostream>

using namespace std;

class Employee
{
private:
    int id;
    string name, address;

public:
    void get_data(void)
    {
        cout << "Enter ID No: "; cin >> id;
        cout << "Enter Name: "; cin >> name;
        cout << "Enter Address: "; cin >> address;
    }
    void print_data(void)
    {
        cout << "The ID is: " << id << endl;
        cout << "The Name is: " << name << endl;
        cout << "The Address is: " << address << endl;
    }
};
```

```
int main()
{
    Employee emp1;

    // Private Members
    // cannot be accessed
    emp1.id = 10;
    emp1.name = "Tingu";

    return 0;
}
```


003_example.cpp

```
#include <iostream>

using namespace std;

class Employee
{
private:
    int id;

public:
    string name, address;

    void get_data(void)
    {
        cout << "Enter ID No: "; cin >> id;
        cout << "Enter Name: "; cin >> name;
        cout << "Enter Address: "; cin >> address;
    }
    void print_data(void)
    {
        cout << "The ID is: " << id << endl;
        cout << "The Name is: " << name << endl;
        cout << "The Address is: " << address << endl;
    }
};
```

```
int main()
{
    Employee emp1;

    // Allowed
    emp1.name = "Tingu";

    return 0;
}
```

C++

Class vs Structure

004_example.cpp

```
#include <iostream>

using namespace std;

struct sEmployee
{
    int id;
    string name;
    string address;
};

class cEmployee
{
    int id;
    string name;
    string address;
};
```

```
int main()
{
    sEmployee emp1;
    cEmployee emp2;

    // Allowed, Since public by
    // default
    emp1.name = "Tingu";
    // Not allowed, Since private
    // by default
    emp2.name = "Pingu";

    return 0;
}
```

005_example.cpp

```
#include <iostream>

using namespace std;

struct sEmployee
{
    int id;
private:
    string name;
    string address;
};

class cEmployee
{
    int id;
public:
    string name;
    string address;
};
```

```
int main()
{
    sEmployee emp1;
    cEmployee emp2;

    // Not allowed, Since declared
    // private
    emp1.name = "Tingu";
    // Allowed, Since declared
    // public
    emp2.name = "Pingu";

    return 0;
}
```

006_example.cpp

```
#include <iostream>

using namespace std;

struct sEmployee
{
    int id;
    string name;
    string address;
};

class cEmployee
{
    int id;
    string name;
    string address;
};
```

```
int main()
{
    sEmployee emp1;
    cEmployee emp2;

    cout << "sizeof emp1 is " << sizeof(emp1) << endl;
    cout << "sizeof emp2 is " << sizeof(emp2) << endl;

    return 0;
}
```

Constructors and Destructors



- Constructor is a special member function of a class that initializes the object of the class.
- The Compiler generates the code such a way that when an object is created a Constructor is called.
- Constructors initialize values to object members after storage is allocated to the object.
- How constructors are different from a normal member function?
 - Constructor has same name as the class itself
 - Constructors don't have return type
 - If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).

C++

Class - Constructors

007_example.cpp

```
#include <iostream>
#include <cstring>

using namespace std;


class Employee
{
public:
    int id;
    char *name;

    Employee()
    {
        id = 0;
        name = (char *)malloc(sizeof(char)*10);
    }
};
```

```
int main()
{
    Employee emp1;

    cout << "The ID is " << emp1.id << endl;
    strcpy(emp1.name, "Tingu");
    cout << "The Name is " << emp1.name << endl;

    return 0;
}
```



C++


Class - Constructors



Constructors are of three types:

- Default Constructor
- Parametrized Constructor
- Copy Constructor

- Default constructors are the one which will have same name as class but doesn't have any arguments.
- A default constructor is so important for initialization of object members, that even if we do not define a constructor explicitly, the compiler will provide a default constructor implicitly.
- The compiler defined default constructors will not have any body.
- Refer slide 31 for default constructor example.



C++

Constructors - Parameterized



- These are the constructors with parameters.
- Using this Constructor you can provide different values to data members of different objects, by passing the appropriate values as argument.



C++

Constructors - Parameterized



008_example.cpp

```
#include <iostream>
#include <cstring>

using namespace std;

class Employee
{
public:
    int id;
    char *name;

    Employee(int x, char *s);


};

Employee::Employee(int x, char *s)
{
    id = x;
    name = (char *)malloc(sizeof(char)*10);
    strcpy(name, s);
}
```

```
int main()
{
    Employee e(10, (char *) "Tingu");

    cout << "The ID is " << e.id << endl;
    cout << "The Name is " << e.name << endl;

    return 0;
}
```



C++

Constructors - Copy



- A copy constructor is a member function which initializes an object using another object of the same class.
- C++ compiler will have one in-built copy constructor, it will be get called whenever an existing object is copied to the new object.



C++

Constructors - Copy



```
#include <iostream>
#include <cstring>

using namespace std;

class Employee
{
public:
    int id;
    char *name;

    Employee(int x, char *s);

};

Employee::Employee(int x, char *s)
{
    id = x;
    name = (char *)malloc(sizeof(char)*10);
    strcpy(name, s);
}
```

```
int main()
{
    Employee e(10, (char *) "Tingu");
    Employee e1 = e;
    //copy constructor called by compiler

    cout << "The ID is " << e.id << endl;
    cout << "The Name is " << e.name << endl;

    cout << "The ID is " << e1.id << endl;
    cout << "The Name is " << e1.name << endl;

    return 0;
}
```

C++

Constructors - Overloading



```
#include <iostream>
using namespace std;

class construct
{
public:
    float area;

    // Constructor with no parameters
    construct()
    {
        cout<<"Constructor with zero args\n";
        area = 0;
    }

    // Constructor with two parameters
    construct(int a, int b)
    {
        cout<<"Constructor with two args\n";
        area = a * b;
    }

    void disp()
    {
        cout<< area<< endl;
    }
};
```

```
int main()
{
    // Constructor Overloading
    // with two different constructors
    // of class name
    construct o;
    construct o2( 10, 20);

    o.disp();
    o2.disp();
    return 1;
}
```

- C++ destructor is a special member function that is executed automatically when an object goes out of the scope or no longer needed.
- C++ destructors are used to de-allocate the memory that has been allocated for the object by the constructor.
- Its syntax is same as constructor except the fact that it is preceded by the tilde sign.

```
//syntax of destructor  
  
~class_name() {  
    .....  
};
```

C++

Class - Destructor

010_example.cpp

```
#include <iostream>
#include <cstring>

using namespace std;

class Employee
{
public:
    int id;
    char *name;

    Employee(int id);
    Employee(int id, char *s);
    ~Employee(void);
};

Employee::Employee(int i, char *s)
{
    id = i;
    name = (char *)malloc(sizeof(char)*10);
    strcpy(name, s);
}


Employee::Employee(int i)
{
    id = i;
}
```

```
Employee::~~Employee(void)
{
    free(name);
}

int main()
{
    Employee e1(10), e2(11, (char *) "Tingu");

    cout << "ID: " << e1.id << endl;
    cout << "Name: " << e1.name << endl;
    cout << "ID: " << e2.id << endl;
    cout << "Name: " << e2.name << endl;

    return 0;
}
```

C++

new and delete operator



- C uses malloc() and calloc() function to allocate memory dynamically at run time and uses free() function to free dynamically allocated memory.
- C++ supports these functions and also has two operators new and delete that perform the task of allocating and freeing the memory in a better and easier way.
- The new operator denotes a request for memory allocation on the Heap.
- If sufficient memory is available, new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.



Syntax to use new operator:

- To allocate memory of any data type, the syntax is:

`Data-type pointer-variable = new data-type;`

- Here, pointer-variable is the pointer of type data-type.
- Data-type could be any built-in data type including array or any user defined data types including structure and class.

Example:

```
// Pointer initialized with NULL, Then request memory for the variable
```

```
int *p = NULL;
```

```
p = new int;
```

OR

```
// Combine declaration of pointer and their assignment
```

```
int *p = new int;
```

C++

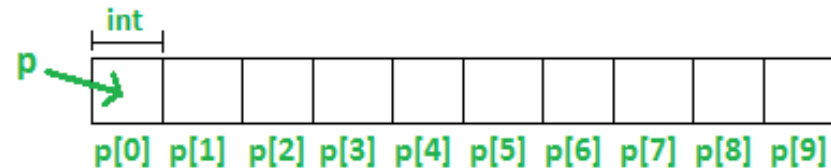
new and delete operator

Example:


```
// Pointer with size  
int *p = new int[10];
```

```
//In c++, we can also initialize the pointer variable  
int *ptr = new int(10);
```

- Dynamically allocates memory for 10 integers continuously of type int and returns pointer to the first element of the sequence, which is assigned to p(a pointer). p[0] refers to first element, p[1] refers to second element and so on.



- We can also initialize the memory using new operator:



C++

new and delete operator



Syntax to use delete operator:

- To de-allocate the dynamically allocated memory, the syntax is:

`Delete pointer_variable_name;`

Example:

```
int *p = NULL;  
p = new int;  
delete p;
```

C++

new and delete with constructor and destructor

007_example.cpp

```
#include <iostream>
#include <cstring>

using namespace std;

class Employee
{
public:
    int id;
    char *name;

    Employee()
    {
        id = 0;
        name = new char[10];
    }
    ~Employee()
    {
        delete name;
    }
};
```


```
int main()
{
    Employee emp1;

    cout << "The ID is " << emp1.id << endl;
    strcpy(emp1.name, "Tingu");
    cout << "The Name is " << emp1.name << endl;

    return 0;
}
```

The Pillars of OOPs





C++

Pillars of OOPs



- What make us to select the C++ as a choice is, its features like
 - Encapsulation
 - Abstraction
 - Inheritance
 - Polymorphism

Encapsulation



- Encapsulation is the process of binding the data and functions together in a single entity or unit.
- In OOPs, encapsulation is achieved through a class.
- Encapsulation leads to an important topic of data hiding / abstraction.

012_example.cpp

```
#include <iostream>

using namespace std;

class Employee
{
    int id; // This member can be accessed with only get and set function since its
           // private, hence we can say its encapsulated
public:
    int get_id(void) { // Getter
        return id;
    }
    void set_id(int id) { // Setter
        this->id = id;
    }
};


int main()
{
    Employee e;

    e.set_id(10);
    cout << "The ID is " << e.get_id() << endl;

    return 0;
}
```

Abstraction





C++

Pillars of OOPs - Abstraction



- Show only “relevant” data and “hide” unnecessary details of an object from the user
 - Focus on the essentials
 - Ignore irrelevant and the unimportant

C++

Abstraction

011_example.cpp

```
#include <iostream>
#include <cstring>

using namespace std;

class Employee
{
private: // Abstracted the members
    int id;
    char *name;
public:
    Employee(int id, char *s);
    ~Employee(void);
    int get_id(void);
    char *get_name(void);
};

int Employee::get_id(void)
{
    return id;
}

char *Employee::get_name(void)
{
    return name;
}
```

```
Employee::Employee(int i, char *s)
{
    id = i;
    name = new char [10];
    strcpy(name, s);
}

Employee::~Employee(void)
{
    delete name;
}

int main()
{
    Employee e1(11, (char *) "Tingu");

    // Cannot access the members directly
    // This, we had seen in the initial
    // examples, Please refer 002_example.cpp

    cout << "ID: " << e1.get_id() << endl;
    cout << "Name: " << e1.get_name() << endl;

    return 0;
}
```

Inheritance

C++

Pillars of OOPs -Inheritance

- Inheritance is a process of creating a new class from an existing class.
- Here, existing class is called as BASE class/PARENT class and new class is called as DERIVED class/CHILD class.
- The derived class inherits all the features from the base class and can have additional features of its own.



I have got
cheese and
meat from him

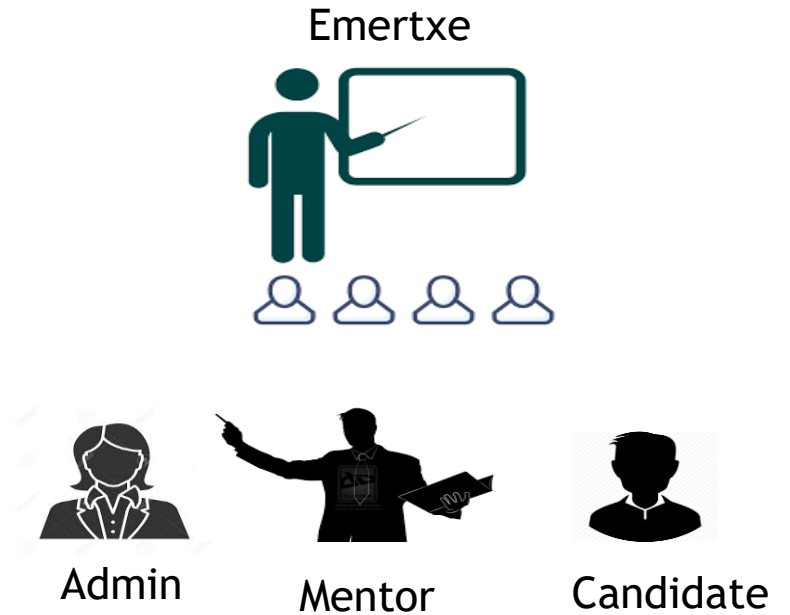


I am just like
him,
except I have
lentils on me :)

C++

Inheritance

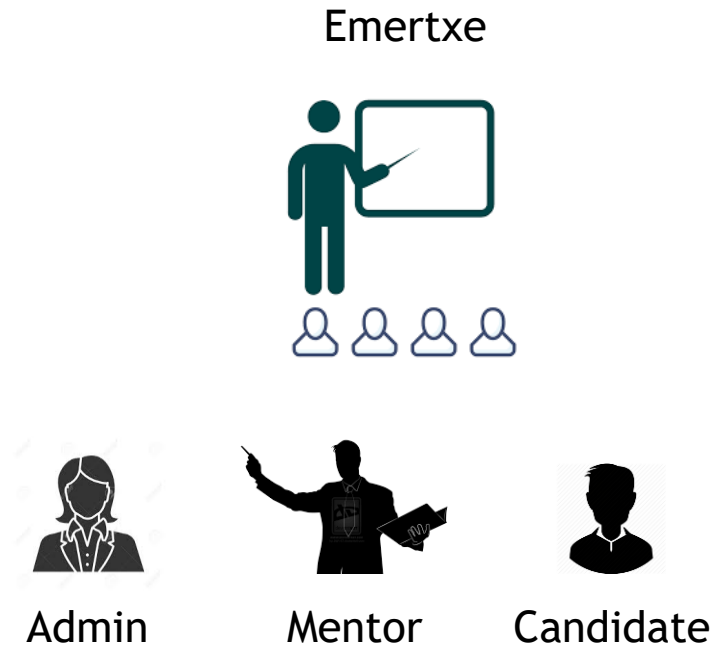
- Now if we take an Organization as an example, Say “**Emertxe**”, We have different group of people here!.
- So if we have a Base Class called **EmertxeMembers**, Some members will be distinct from other some ways
- So the Mentors, Candidates will have a Sub Class along with the Base Class with their specifics



C++

Inheritance

- So what identity, attribute and behaviors do people have in common?



Attributes

- Name
- ID
- Address

Behaviors

- Display Profile
- Change Profile

- Let see some things specific to Candidate apart from the main attributes and behaviors



Candidate

Inherit Base Class

- Name
- ID
- Address
- Display Profile
- Change Profile

Sub Class

- Batch ID
- Course Taken
- Current Module
- Grade
- Change Course
- Add a Class Taken

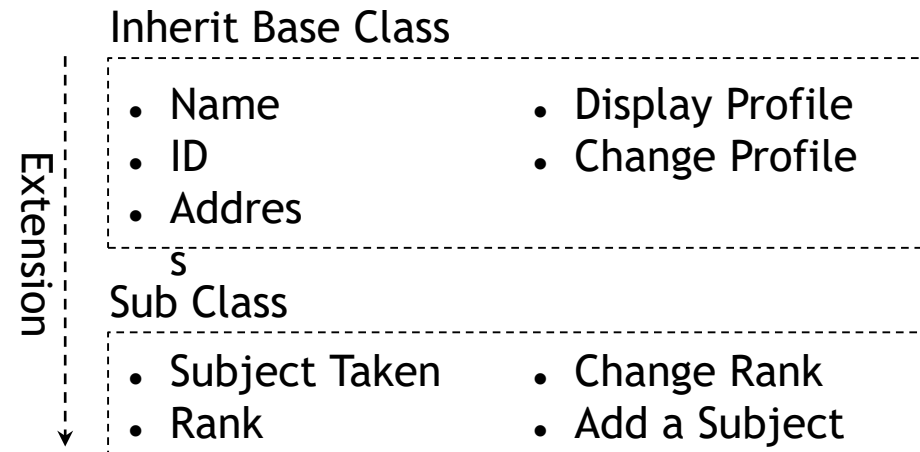
Extension



- On the same line the Mentor may have some extra features as shown



Mentor



```
class DERIVED_Class_name : access_modifiers BASE_class_name
{
    /* derived class members */
};
```

```
class Parent
{
    int id;
    float f;
public:
    Parent()
    {
        id=0;
        f=10.05;
    }
    void display()
    {
        cout<<id<<endl<<f<<endl;
    }
};
```

```
class Child:public Parent
{
};
```

```
int main()
{
    Parent p;
    cout<<sizeof(p)<<endl;
    Child c;
    cout<<sizeof(c)<<endl;
}
```

C++

Inheritance - Protected



```
class Parent
{
    int id;
    protected:
        float f;

    public:
        Parent()
        {
            id=0;
            f=10.05;
        }
        void display()
        {
            cout<<id<<endl<<f<<endl;
        }
};
```

```
class Child:public Parent
{
    public:
        void ch()
        {
            cout<<f<<endl;
        }
};
```

```
int main()
{
    Parent p;
    p.display();
    Child c;
    c.ch();
    return 0;
}
```

C++

Inheritance - Constructor Execution

```
class Parent
{
    int id;
    float f;
public:
    Parent()
    {
        cout<<"Parent Constructor\n"
        id=0;
        f=10.05;
    }
    void display()
    {
        cout<<id<<endl<<f<<endl;
    }
};
```

```
class Child:public Parent
{
public:
    Child()
    {
        cout<<"Child Constructor\n";
    }
};
```

```
int main()
{
    Parent p;
    cout<<sizeof(p)<<endl;
    Child c;
    cout<<sizeof(c)<<endl;
}
```

Public inheritance

- Public inheritance is by far the most commonly used type of inheritance.
- Fortunately, public inheritance is also the easiest to understand.
- When you inherit a base class publicly, inherited public members stay public
- Inherited protected members stay protected.
- Inherited private members, which were inaccessible because they were private in the base class, stay inaccessible.

Access specifier in base class	Access specifier when inherited publicly
Public	Public
Private	Inaccessible
Protected	Protected

C++

Inheritance - Public Inheritance

```
class Base
{
public:
    int m_public;
private:
    int m_private;
protected:
    int m_protected;
};

class Pub: public Base // note: public inheritance
{
public:
    Pub()
    {
        m_public = 1;
        // okay: m_public was inherited as public
        m_private = 2;
        // not okay: m_private is inaccessible from derived class
        m_protected = 3;
        // okay: m_protected was inherited as protected
    }
};
```

```
int main()
{
    Base base;
    base.m_public = 1;
    // okay: m_public is public in Base
    base.m_private = 2;
    // not okay: m_private is private in Base
    base.m_protected = 3;
    // not okay: m_protected is protected in Base

    Pub pub;
    pub.m_public = 1;
    // okay: m_public is public in Pub
    pub.m_private = 2;
    // not okay: m_private is inaccessible in Pub
    pub.m_protected = 3;
    // not okay: m_protected is protected in Pub

    return 0;
}
```


Private inheritance

- With private inheritance, all members from the base class are inherited as private.
- This means private members stay private, and protected and public members become private.

Access specifier in base class	Access specifier when inherited privately
Public	Private
Private	Inaccessible
Protected	Private

C++

Inheritance - Private Inheritance

```
class Base
{
public:
    int m_public;
private:
    int m_private;
protected:
    int m_protected;
};

class Pri: private Base // note: private inheritance
{
public:
    Pri()
    {
        m_public = 1;
        // okay: m_public is now private in Pri
        m_private = 2;
        // not okay:
        m_protected = 3;
        // okay: m_protected is now private in Pri
    }
};
```

```
int main()
{
    Base base;
    base.m_public = 1;
    // okay: m_public is public in Base
    base.m_private = 2;
    // not okay: m_private is private in Base
    base.m_protected = 3;
    // not okay: m_protected is protected in Base

    Pri pri;
    pri.m_public = 1;
    // not okay: m_public is private in Pri
    pri.m_private = 2;
    // not okay: m_private is inaccessible in Pri
    pri.m_protected = 3;
    // not okay: m_protected is private in Pri

    return 0;
}
```

Protected inheritance

- Protected inheritance is the last method of inheritance.
- It is almost never used, except in very particular cases.
- With protected inheritance, the public and protected members become protected, and private members stay inaccessible.

Access specifier in base class	Access specifier when inherited protectedly
Public	Protected
Private	Inaccessible
Protected	Protected

C++

Inheritance - Protected Inheritance


```
class Base
{
public:
    int m_public;
private:
    int m_private;
protected:
    int m_protected;
};

class Pro: protected Base // note: private inheritance
{
public:
    Pro()
    {
        m_public = 1;
        // okay: m_public is now protected in Pro
        m_private = 2;
        // not okay:
        m_protected = 3;
        // okay: m_protected is now protected in Pro
    }
};
```

```
int main()
{
    Base base;
    base.m_public = 1;
    // okay: m_public is public in Base
    base.m_private = 2;
    // not okay: m_private is private in Base
    base.m_protected = 3;
    // not okay: m_protected is protected in Base

    Pro pro;
    pro.m_public = 1;
    // not okay: m_public is protected in Pro
    pro.m_private = 2;
    // not okay: m_private is inaccessible in Pro
    pro.m_protected = 3;
    // not okay: m_protected is protected in Pro

    return 0;
}
```



C++

Inheritance - Types



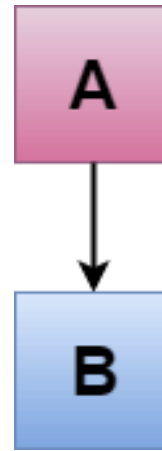
- In C++, we have following types of inheritance:
 - Single Inheritance
 - Multilevel Inheritance
 - Multiple Inheritance
 - Hierarchical Inheritance
 - Hybrid Inheritance

C++

Inheritance Types - Single Inheritance



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



- Where 'A' is the base class, and 'B' is the derived class.



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



- Where 'A' is the base class, and 'B' is both derived class and BASE class, 'C' is derived class.

C++

Inheritance - Multilevel Inheritance

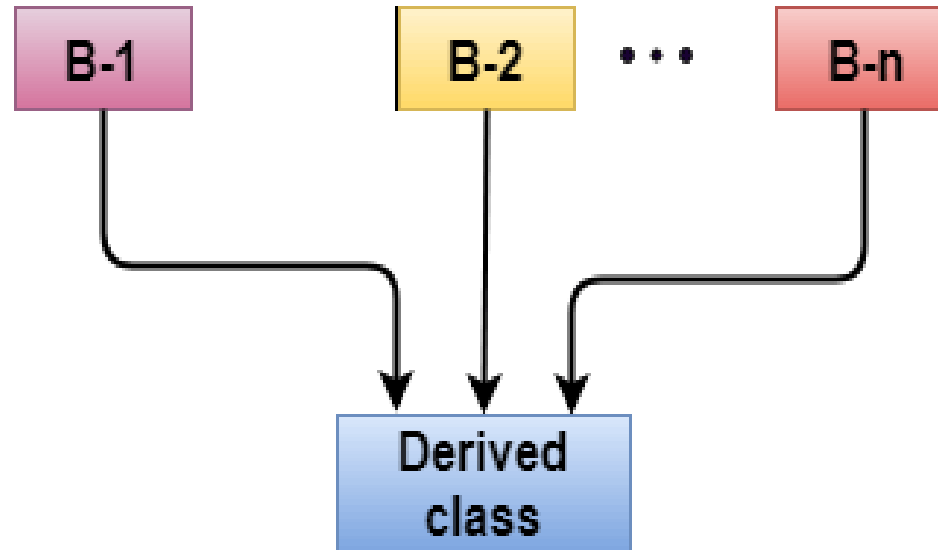
```
class A
{
public:
    A()
    {
        cout<<"A's Constructor\n";
    }
};

class B: public A
{
public:
    B()
    {
        cout<<"B's Constructor\n";
    }
};

class C: public B
{
public:
    C()
    {
        cout<<"C's Constructor\n";
    }
};
```

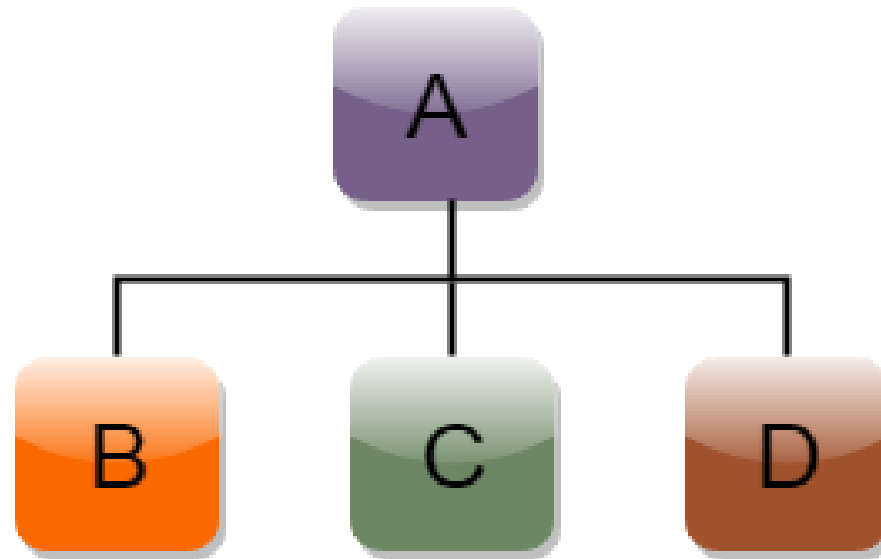
```
int main()
{
    A a;
    B b;
    C c;
}
```


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



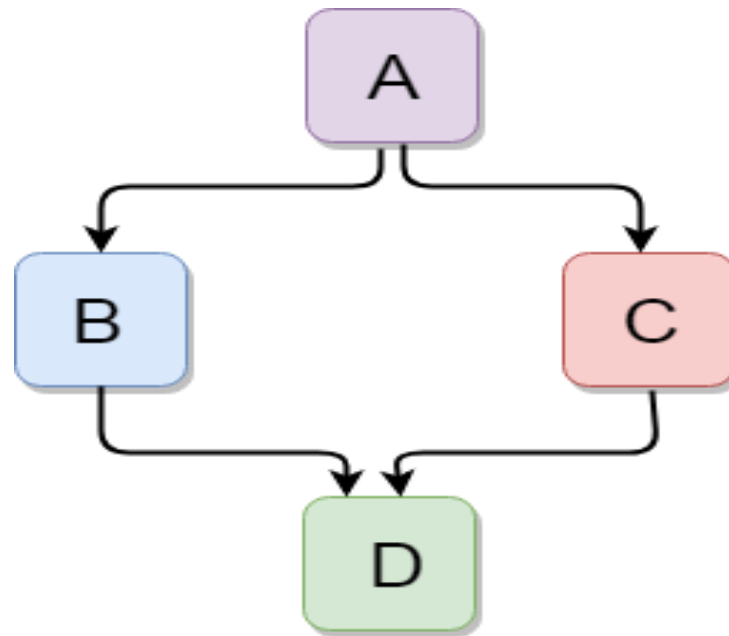


- Hierarchical inheritance is defined as the process of deriving more than one class from a base class.





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



- Then, the base class of Emertxe Members would look like

013_example.h

```
#ifndef EXAMPLE_013_H
#define EXAMPLE_013_H

#include <iostream>
#include <cstring>

using namespace std;

class EmertxeMember
{
protected:
    int id;
    string name;
    string address;
public:
    EmertxeMember(int id, string n, string a)
    {
        this->id = id;
        name = n;
        address = a;
    }
    void display_profile(void);
};
```



- The extended candidate class would look like this

013_example.h

```
class Candidate : public EmertxeMember
{
    // Note have not considered all cases said in the previous slide
    string course;
    int year;
public:
    Candidate(int id, string n, string a, string course, int year);
    void display_profile(void);
};
```

C++

Inheritance

- The extended mentor class would look like this

013_example.h

```
class Mentor : public EmertxeMember
{
    // Note have not considered all cases said in the previous slide
    string sub_taught;
    string rank;
public:
    Mentor(int id, string n, string a, string sub_taught, string year);
    void display_profile(void);
};

#endif
```



- The definition of all the methods can be put as

013_example.cpp

```
#include "013_example.h"

Mentor::Mentor(int id, string n, string a, string sub_taught, string rank)
    :EmertxeMember(id, n, a)
{
    this->sub_taught = sub_taught;
    this->rank = rank;
}

Candidate::Candidate(int id, string n, string a, string course, int year)
    :EmertxeMember(id, n, a)
{
    this->course = course;
    this->year = year;
}
```

C++

Inheritance

013_example.cpp

```
void EmertxeMember::display_profile(void)
{
    cout << "ID: " << id << endl;
    cout << "Name: " << name << endl;
    cout << "Address: " << address << endl;
}

void Mentor::display_profile(void) // Override the base class definition
{
    cout << "ID: " << id << endl;
    cout << "Name: " << name << endl;
    cout << "Address: " << address << endl;
    cout << "Subject Taught: " << sub_taught << endl;
    cout << "Rank: " << rank << endl;
}

void Candidate::display_profile(void) // Override the base class definition
{
    cout << "ID: " << id << endl;
    cout << "Name: " << name << endl;
    cout << "Address: " << address << endl;
    cout << "Course: " << course << endl;
    cout << "Year: " << year << endl;
}
```


C++

Inheritance

014_example.cpp

```
#include "013_example.h"

// Please compile along with 013_example.cpp

int main()
{
    EmertxeMember m1(100, "Ringu", "Mangalore");
    Mentor m2(108, "Tingu", "Mysore", "Linux Systems", "Senior");
    Candidate c1(120, "Pingu", "Bangalore", "ECEP", 2019);

    cout << "m1:--->\n"; m1.display_profile();
    cout << "m2:--->\n"; m2.display_profile();
    cout << "c1:--->\n"; c1.display_profile();

    return 0;
}
```

C++

Constructor Invocation Explicitly

```
#include <iostream>
using namespace std;
class Polygon {
    protected:
        int width, height;
    public:
        Polygon(int a, int b)
        {
            width=a; height=b;
        }
        display_value()
        {
            cout<<width<<" "<<height<<" "<<endl;
        }
};
```

```
int main()
{
    Polygon p = Polygon(4,40);
    p.display_value();
    return 0;
}
```

C++

Dynamic Object Creation - new

```
#include <iostream>
using namespace std;
class Polygon {
protected:
    int width, height;
public:
    Polygon(int a, int b)
    {
        cout<<"Constructor called\n";
        width=a; height=b;
    }
    display_value()
    {
        cout<<width<<" "<<height<<" "<<endl;
    }
};
```

```
int main()
{
    Polygon *p = new Polygon(4,40);
    p->display_value();
    return 0;
}
```

C++

Dynamic Object Creation - malloc()

```
#include <iostream>
using namespace std;
class Polygon {
    protected:
        int width, height;
    public:
        Polygon(int a, int b)
        {
            cout<<"Constructor called\n";
            width=a; height=b;
        }
        display_value()
        {
            cout<<width<<" "<<height<<" "<<endl;
        }
};
```

```
int main()
{
    Polygon *p = (Polygon *) malloc(sizeof(Polygon));
    p->display_value();
    return 0;
}
```

C++

Pointer to base class

```
// pointers to base class
#include <iostream>
using namespace std;
class Polygon {
    protected:
    int width,
    height;
    public:
    void set_values (int a, int b)
    {
        width=a; height=b;
    }
};
class Rectangle: public Polygon {
    public:
    int area()
    {
        return width*height;
    }
};
```

```
class Triangle: public Polygon {
    public:
    int area()
    {
        return width*height/2;
    }
};
int main ()
{
    Rectangle rect;
    Triangle trgl;
    Polygon * ppoly1 = &rect;
    Polygon * ppoly2 = &trgl;
    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    cout << rect.area() << '\n';
    cout << trgl.area() << '\n';
    return 0;
}
```

Polymorphism





C++

Polymorphism

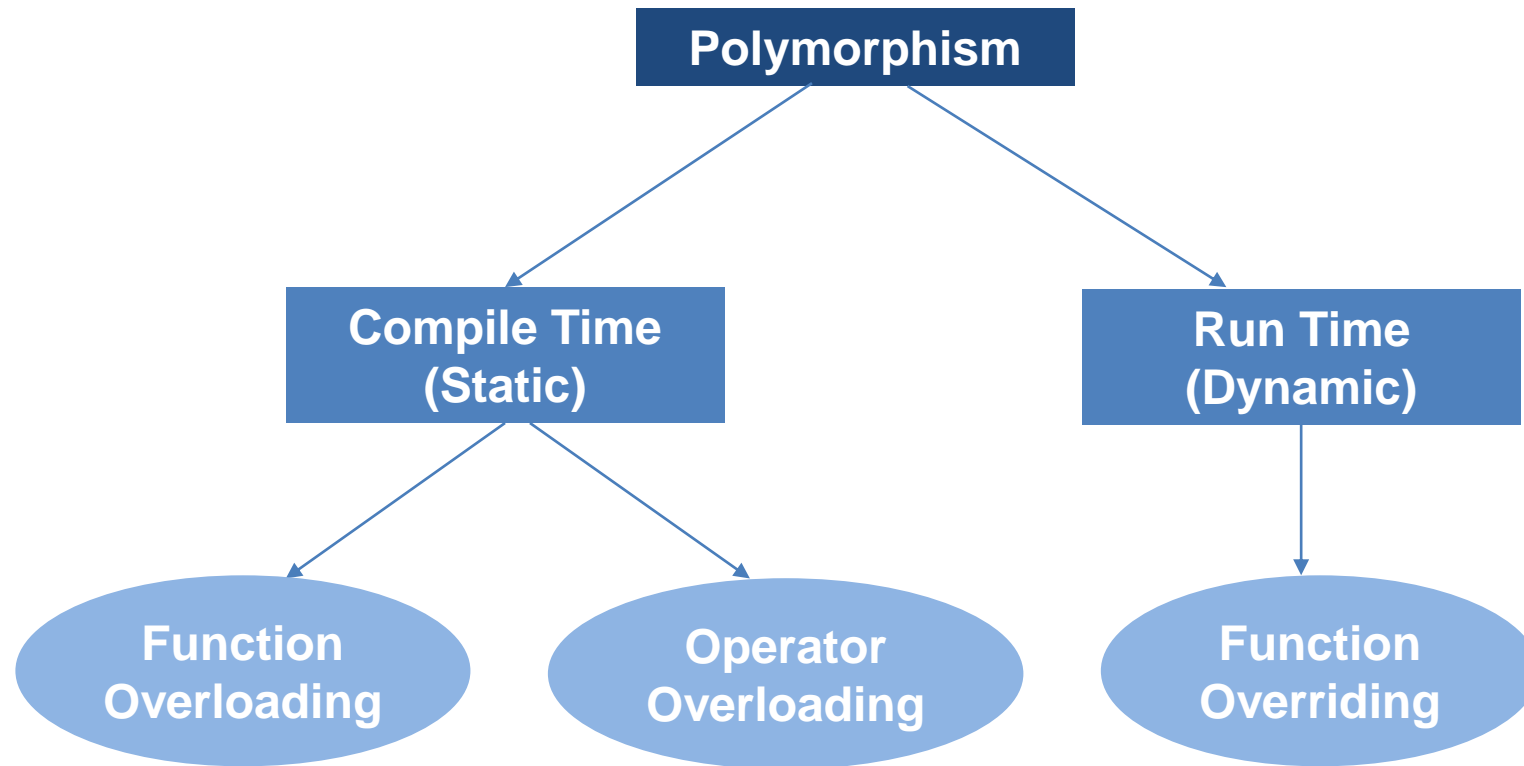


- The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms.
- Polymorphism in C++ is basically the ability for data to be processed or represented in more than one form.
- Let's consider a real-life example of polymorphism. A lady behaves like a teacher in a classroom, mother or daughter in a home and customer in a market. Here, a single person is behaving differently according to the situations.



C++

Polymorphism - Types



- Function overloading in C++ is when two or more function has similar names but have different parameters.
- Parameters can be different at times, and it can be the different return type of the function, the number of arguments in the function.
- We cannot only overload the function only the basis of return type at declaration. Merely changing the return type won't overload the function.



```
void function1 (int a)
{
    cout<<a<<endl;
}

void function1 (float b)
{
    cout<<b<<endl;
}

float function1 (int a, float b)
{
    return a+b;
}
```

C++

Polymorphism - Function Overloading

009_example.cpp

```
#include <iostream>

using namespace std;

// Function Overloading
int add(int n1, int n2)
{
    return n1 + n2;
}

double add(double n1, double n2)
{
    return n1 + n2;
}

string add(string s1, string s2)
{
    // Operator Overloading
    return s1 + s2;
}
```

```
int main()
{
    cout << add(5, 10) << endl;
    cout << add(3.5, 6.5) << endl;
    cout << add("Hell", "o") << endl;


    return 0;
}
```

C++

Polymorphism - Function overloading with class

```
#include <iostream>
using namespace std;
class FunctionOverloading
{
public:
    void calculation(int a, int b, int k)
        /*addition of two numbers*/
    {
        cout << "The sum is = " << a + b + k << endl;
    }
    void calculation(double c, double d, double e)
        /*Multiplication of two numbers*/
    {
        cout << "Multiplication is = " << c * d * e << endl;
    }
    int calculation(int f, int g)
        /*Division of two numbers*/
    {
        return (f / g);
    }
};
```

```
int main()
{
    FunctionOverloading obj1;
    obj1.calculation(5, 6, 7);
    obj1.calculation(2.34, 4.54, 6.72);
    cout << "The division is= " << obj1.calculation(10, 5) << endl;
}
```



C++

Polymorphism - Operator Overloading



- Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type.
- Operator overloading is used to overload or redefines most of the operators available in C++.
- For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.
- Operator that cannot be overloaded are as follows:
 - ☐ Scope operator (::)
 - ☐ Sizeof
 - ☐ member selector(.)
 - ☐ member pointer selector(*)
 - ☐ ternary operator(?:)



- The unary operators operate on a single operand and following are the examples of Unary operators –
 - ❖ The increment (++) and decrement (--) operators.
 - ❖ The unary minus (-) operator.
 - ❖ The logical not (!) operator.
- The unary operators operate on the object for which they were called and normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj but sometime they can be used as postfix as well like obj++ or obj--.

C++

Polymorphism - Operator Overloading (Unary Operator)

```
#include <iostream>
using namespace std;
class Distance
{
    private:
        int feet; // 0 to infinite
        int inches; // 0 to 12
    public:
        // required constructors
        Distance(int f=0, int i=0):feet(f),inches(i)
        {
        }

        // method to display distance
        void displayDistance()
        {
            cout << "F: " << feet << " I:" << inches << endl;
        }

        // overloaded minus (-) operator
        void operator- ()
        {
            feet = -feet;
            inches = -inches;
        }
};
```

```
int main()
{
    Distance D1(11, 10), D2(-5, 11);
    -D1; // apply negation
    D1.displayDistance();//display D1
    -D2; // apply negation
    D2.displayDistance();//display D2
    return 0;
}
```

C++

Polymorphism - Unary Operator Overloading(Prefix and Postfix)

```
#include <iostream>
using namespace std;
Class Unary
{
    private:
        int i;
        int i1;
    public:
        // required constructors
        Unary(int f=0, int i=0):i(f),i1(i)
        {
        }

        void displayDistance()
        {
            cout << "i: " << i << " i1:" << i1 << endl;
        }

        // overloaded pre increment operator(prefix)
        void operator ++()
        {
            i+=2;
            i1+=2;
        }
};
```

```
// overloaded pre increment operator(postfix)
void operator ++(int)
{
    i+=2;
    i1+=2;
}

};

int main()
{
    Unary U1(11, 10), U2(-5, 11);
    ++U1; //pre increment
    U1.displayDistance();//display U1
    U1++; //post increment
    U1.displayDistance();//display U1

    U2++; //post increment
    U2.displayDistance();//display U2
    return 0;
}
```




- The binary operators operate on two operands and following are the examples of Binary operators – arithmetic, relational, logical, and assignment.
- The Binary operators operate on the left object on which operator has been called. The right operand should be passed as argument to the operator overloading function.
- Example - `Object1+10`, `Object2+Object2`, `Object1+10.50....`

C++

Polymorphism - Binary Operator

```
#include <iostream>
using namespace std;
Class Binary
{
    private:
        int i;
        int i1;
    public:
        // required constructors
        Binary(int f=0, int i=0):i(f),i1(i)
        {
        }


        void display()
        {
            cout << "i: " << i << " i1:" << i1 << endl;
        }

        void operator +(int a)
        {
            i+=a;
            i1+=a;
        }
};
```

```
void operator +(Binary b)
{
    i+=b.i;
    i1+=b.i1;
}

};

int main()
{
    Binary B1(11, 10), B2(-5, 11);
    int a=10;
    B1+a;
    B1.displayDistance();
    B1+B2;
    B1.displayDistance();
    B2.displayDistance();
    return 0;
}
```



C++

Polymorphism - Overriding



- Function overriding is a feature that allows us to have a same function in child class which is already present in the parent class.
- A child class inherits the data members and member functions of parent class, but when you want to override a functionality in the child class then you can use function overriding.
- It is like creating a new version of an old function, in the child class.
- To override a function you must have the same signature in child class.
- By signature, means the data type and sequence of parameters. If we don't have any parameter in the parent function then in child function, you cannot have any arguments. And also return type also should be exactly same as parent function.

C++

Polymorphism - Overriding

```
#include <iostream>
using namespace std;
class BaseClass
{
    public:
        void disp()
        {
            cout<<"Function of Parent Class";
        }
};
class DerivedClass: public BaseClass
{
    public:
        void disp()
        {
            cout<<"Function of Child Class";
        }
};
```

```
int main()
{
    DerivedClass obj = DerivedClass();
    obj.disp();
    return 0;
}
```

C++

Polymorphism - Overriding

```
#include <iostream>
using namespace std;
class BaseClass
{
    public:
        void disp()
        {
            cout<<"Function of Parent Class";
        }
};
class DerivedClass: public BaseClass
{
    public:
        void disp()
        {
            cout<<"Function of Child Class";
        }
};
```

```
int main()
{
    BaseClass *obj = new DerivedClass();
    obj->disp();
    return 0;
}
```

- So ability of one existing type, say X to appear as and be used like another type Y e.g.
 - Candidate object can be used in place of an EmertxeMember object

015_example.cpp

```
#include "013_example.h"

// Please compile along with 013_example.cpp

int main()
{
    EmertxeMember *m1 = new EmertxeMember(200, "Tingu", "Mysore");
    EmertxeMember *m2 = new Candidate(300, "Pingu", "Bangalore", "ECEP", 2019);

    cout << "m1:--->\n"; m1->display_profile();
    cout << "m2:--->\n"; m2->display_profile();

    return 0;
}
```

- Did you observe the types used above?!!



- In the previous example object m1 point to the **declared type** which is the **actual type**!
- But in case of m2 the declared type is EmertxeMember and it points to Candidate, which is a actual type!
- So every object has a type declared at compilation time, but at run time it may point to the actual type



- So if we execute the previous code, the result is as follows

```
user@user:~] g++ 014_example.cpp 013_example.cpp
user@user:~] ./a.out
m1:-->
ID: 200
Name: Tingu
Address: Mysore
m2:-->
ID: 300
Name: Pingu
Address: Bangalore
user@user:~]
```


Note that, the
course name and *year*
is *missing* in the output!!
Why??

- This is not complete even though we called the override function!!
- Here comes the need of virtual function concepts



- Member function that you expect to be redefined (i.e is overridden) in derived classes
- So if we refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function
- A base class function call is fixed before the program is executed. This is called as **early binding** or **static linkage** because the these functions is set during the compilation of the program.





C++

Virtual Functions



- Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function. This sort of operation is referred to as **dynamic linkage**, or **late binding**.
- This concepts are mainly used to achieve **Runtime Polymorphism**



C++

Polymorphism - Overriding - Virtual Function

```
#include <iostream>
using namespace std;
class BaseClass
{
    public:
        virtual void disp()
        {
            cout<<"Function of Parent Class";
        }
};
class DerivedClass: public BaseClass
{
    public:
        void disp()
        {
            cout<<"Function of Child Class";
        }
};
```

```
int main()
{
    BaseClass *obj = new DerivedClass();
    obj->disp();
    return 0;
}
```

- Now lets modify 013_example.h as shown here, and understand the Runtime Polymorphism

016_example.h

```
#ifndef EXAMPLE_016_H
#define EXAMPLE_016_H

#include <iostream>
#include <cstring>

using namespace std;

class EmertxeMember
{
protected:
    int id;
    string name;
    string address;
public:
    EmertxeMember(int id, string n, string a)
    {
        this->id = id;
        name = n;
        address = a;
    }
    virtual void display_profile(void);
};
```

016_example.h

```
class Candidate : public EmertxeMember
{
    // Note have not considered all cases said in the previous slide
    int course;
    int year;
public:
    Candidate(int id, string n, string a, int course, int year);
    void display_profile(void);
};

class Mentor : public EmertxeMember
{
    // Note have not considered all cases said in the previous slide
    string sub_taught;
    string rank;
public:
    Mentor(int id, string n, string a, string sub_taught, string year);
    void display_profile(void);
};

#endif
```

C++

Virtual Functions

016_example.cpp

```
#include "016_example.h"

Mentor::Mentor(int id, string n, string a, string sub_taught, string year)
    :EmertxeMember(id, n, a)
{
    this->sub_taught = sub_taught;
    this->rank = rank;
}

Candidate::Candidate(int id, string n, string a, int course, int year)
    :EmertxeMember(id, n, a)
{
    this->course = course;
    this->year = year;
}
```

016_example.cpp

```
void EmertxeMember::display_profile(void)
{
    cout << "ID: " << id << endl;
    cout << "Name: " << name << endl;
    cout << "Address: " << address << endl;
}

void Mentor::display_profile(void) // Override the base class definition
{
    cout << "ID: " << id << endl;
    cout << "Name: " << name << endl;
    cout << "Address: " << address << endl;
    cout << "Subject Taught: " << sub_taught << endl;
    cout << "Rank: " << rank << endl;
}

void Candidate::display_profile(void) // Override the base class definition
{
    cout << "ID: " << id << endl;
    cout << "Name: " << name << endl;
    cout << "Address: " << address << endl;
    cout << "Course: " << course << endl;
    cout << "Year: " << year << endl;
}
```

017_example.cpp

```
#include "016_example.h"


// Please compile along with 016_example.cpp

int main()
{
    EmertxeMember *m1 = new EmertxeMember(200, "Tingu", "Mysore");
    EmertxeMember *m2 = new Candidate(300, "Pingu", "Bangalore", "ECEP", 2019);

    cout << "m1:--->\n"; m1->display_profile();
    cout << "m2:--->\n"; m2->display_profile();

    return 0;
}
```

- You may observe the output of the code now



C++

Virtual Functions



- What is happening behind the scene?
 - A virtual table (V-Table) is created
 - Stores pointers to all virtual functions
 - Created per each class
 - Lookup during the function call

- What about constructors?
 - To create an object, you must know its exact type. The VPTR has not even been initialized at this point, so the answers in **NO**
- Then what about destructors?
 - Yes, we must always clean up the mess created in the subclass (else, we risk memory leaks!)

- If there is no meaningful definition you could give for the function in the base class. But still you want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class

- We may declare it as following

Example

```
class EmertxeMember
{
protected:
    int id;
    string name;
    string address;
public:
    EmertxeMember(int id, string n, string a)
    {
        // ...
    }
    virtual void display_profile(void);

    // Pure Virtual Function
    virtual void change_profile(void) = 0;
};
```

- A class is abstract if it has at least one pure virtual function
- Sometimes you want to inherit only declarations, not definitions
- A method without an implementation is called an abstract method
- Often used to create an interface
- We can have pointers and references of abstract class type
- If we do not override the pure virtual function in derived class, then derived class also becomes abstract class

018_example.cpp

```
#include <iostream>

using namespace std;

class Polygon
{
protected:
    int width, height;
    string shape_name;
public:
    Polygon() { }
    Polygon(int a, int b, string name) : width(a), height(b), shape_name(name) { }

    string get_name(void) {
        return shape_name;
    }

    // A pure virtual functions
    virtual int get_area(void) = 0;

    void print_area(void) {
        cout << "Area of " << this->get_name() << " is "
              << this->get_area() << endl;
    }
};
```

018_example.cpp

```
class Rectangle: public Polygon
{
    public:
        Rectangle(int a, int b, string name) : Polygon(a, b, name) { }

        int get_area(void)
        {
            return width * height;
        }
};

class Triangle: public Polygon
{
    public:
        Triangle(int a, int b, string name) : Polygon(a, b, name) { }

        int get_area(void)
        {
            return width * height / 2;
        }
};
```

018_example.cpp

```
int main()
{
    Rectangle rect (4, 5, "Rectangle");
    Triangle trgl (4, 5, "Triangle");
    Polygon *shapes[] = {&rect, &trgl};

    for (int i = 0; i < 2; i++)
    {
        shapes[i]->print_area();
    }

    return 0;
}
```

- Is a constructor needed? Because, the class will never be instantiated!
 - Yes, The subclass will inherit it, So a constructor will be used initialize its members
- What about destructor? Because, the class will never be created
 - Yes, Always define a virtual destructor in the base class, to make sure that the destructor of its subclass is called!

Properties of C++



- An abstract space that contains a set of names
- Is a declarative region that provides a scope to the identifiers (names of the types, function, variables etc) within it
- Useful for resolving naming conflicts

019_example.cpp

```
#include <iostream>

using namespace std;

int main()
{
    int x = 10;
    cout << x << endl;

    double x = 15.5; // Not allowed to have the same name in a local space!
    cout << x << endl;

    return 0;
}
```

C++

Namespace

020_example.cpp

```
#include <iostream>

using namespace std;

int x = 10;

int main()
{
    double x = 10.5;
    cout << x << endl; // How to get the global x refernce here

    return 0;
}
```

C++

Namespace



021_example.cpp

```
#include <iostream>

using namespace std;

namespace global
{
    int x = 10;
}

int main()
{
    double x = 10.5;
    cout << global::x << endl;

    return 0;
}
```

C++

Namespace

022_example.cpp

```
#include <iostream>

using namespace std;

namespace first
{
    int x = 10;
}

namespace second
{
    double x = 12.120;
}

int main()
{
    double x = 10.5;

    cout << x << endl;
    cout << first::x << endl;
    cout << second::x << endl;

    return 0;
}
```

023_example.cpp

```
#include <iostream>

using namespace std;

namespace first
{
    int x = 10;
}

namespace second
{
    double x = 12.120;
}

int main()
{
    using namespace second;

    cout << x << endl;

    return 0;
}
```

C++

Namespace



024_example.cpp

```
#include <iostream>

using namespace std;

namespace MySpace
{
    class Employee
    {
        public:
            int id;
            string name;
    };
}

class Employee
{
    public:
        int id;
        string name;
};
```

```
int main()
{
    Employee emp1;
    MySpace::Employee emp2;

    emp1.name = "Tingu";
    emp2.name = "Pingu";

    cout << emp1.name << endl;
    cout << emp2.name << endl;

    return 0;
}
```



- Templates are powerful features of C++ which allows you to write generic programs
- Templates are often used in larger code-base for the purpose of code re-usability and flexibility of the programs
- The concept of templates can be used in two different ways
 - Function Templates
 - Class Templates

- Function templates are special functions that can operate with generic types.
- This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.
- The format for declaring function templates with type parameters is:

```
template <class identifier> function_declaration;
```

```
template <typename identifier> function_declaration;
```

- The only difference between both prototypes is the use of either the keyword class or the keyword typename. Its use is indistinct, since both expressions have exactly the same meaning and behave exactly the same way.

C++

Templates

```
#include <iostream>

using namespace std;

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

int main()
{
    cout << Max(10, 20) << endl;
    cout << Max(33.5, 20.2) << endl;
    cout << Max(3.5, 10.2) << endl;
    cout << Max('A', 'B') << endl;
    cout << Max('Z', 'Y') << endl;

    return 0;
}
```



- A class template provides a specification for generating classes based on parameters.
- Class templates are generally used to implement containers.
- A class template is instantiated by passing a given set of types to it as template arguments.

```
template <class T>
class MyTemplate
{
    T element;
    public:
    MyTemplate (T arg)
    {
        element=arg;
    }
    T divideBy2 ()
    {
        return element/2;
    }
};
int main()
{
    MyTemplate <int>m(10);
    cout<<"division: "<<m.divideBy2()<<endl;
    MyTemplate <float>m1(10.50f);
    cout<<"division: "<<m1.divideBy2()<<endl;
    return 0;
}
```

C++ Exception Handling



Exception Handling

- *An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.*
- *Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: try, catch, and throw.*
- *throw – A program throws an exception when a problem shows up. This is done using a throw keyword.*
- *catch – A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.*
- *try – A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.*

```
#include <iostream>
using namespace std;
int main()
{
    int a, b;
    cout << "Enter two values: ";
    cin >> a >> b;
    try
    {
        if (b != 0) {
            cout << "Res: " << a / b << endl;
        }
        else {
            throw b;
        }
    }
    catch(int x)
    {
        cout << "Caught DIVIDE_BY_ZERO ERROR" << "b: " << x << endl;
    }
}
```

Thank You