LESSON 10: Class

Class (1)

Classes are an expanded concept of data structures:

like data structures, they can contain data members (also called properties/attributes), but they can also contain functions as members (also called methods/activities).

class = data + functions

Class (2)

- Properties:
  - Name
  - Age
  - Color
  - Weight
- Activities
  - Eat
  - Sleep
  - Run
  - Bark

```
class Dog {
public:
    void eat();
    void sleep();
    void run();
    void bark();
private:
    string mName;
    int mAge;
    string  mColor;
    int mWeight;
};
```

Object

An ***object*** is an instantiation of a class ==> That means a class is the a data type, and an ***object*** is a variable of this type.

```
Dog dog;  // dog is an object
```

Declare class (1)

```
class <class_name> {
  [access_specifier_1]:
  member1;
  [access_specifier_2]:
  member2;
  ...
};
```

- private
- protected
- public

- data
- function

Declare class (2)

```
class Rectangle {
  int mWidth;
  int mHeight;
public:
  void setValues(int x, int y);
  int getArea(void);
};
```

IMPLEMENT CLASS (1)

```
void Rectangle::setValues(int x, int y) {
  mWidth = x;
  mHeight = y;
}

int Rectangle::getArea() {
  return mWidth*mHeight;
}
```

IMPLEMENT CLASS (2)

Here is the complete example of class Rectangle

Rectangle.h

```
class Rectangle {
   int mWidth;
   int mHeight;

public:
   void setValues(int, int);
   int getArea(void);

};
```

Rectangle.cpp

```
#include "Rectangle.h"

void Rectangle::setValues(int x, int y) {
     mWidth = x;
     mHeight = y;
}

int getArea() {
     return mWidth*mHeight;
}
```

main.cpp

```
#include "Rectangle.h"
#include <iostream>
using namespace std;

int main() {
     Rectangle rect;
     rect.setValue(3, 4);
     cout << "area = " << rect.getArea();
     return 0;
}
```

ACCESS SPECIFIER (1)

Rectangle.h

```
class Rectangle {

public:
   int mWidth;
   int mHeight;

public:
   void setValues(int, int);
   int getArea(void);

};
```

Rectangle.cpp

```
#include "Rectangle.h"

void Rectangle::setValues(int x, int y) {
     mWidth = x;
     mHeight = y;
}

int getArea() {
     return mWidth*mHeight;
}
```

main.cpp

```
#include "Rectangle.h"
#include <iostream>
using namespace std;

int main() {
     Rectangle rect;
     rect.mWidth = 3;
     rect.mHeight = 4;
     cout << "area = " << rect.getArea();
     return 0;
}
```

ACCESS SPECIFIER (2)

Rectangle.h

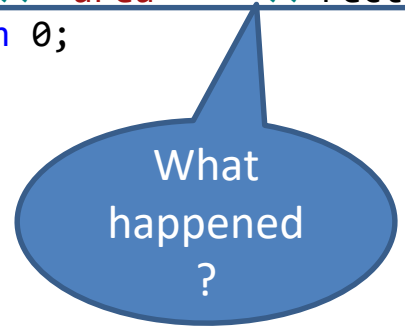
```
class Rectangle {
    int mWidth;
    int mHeight;
public:
    void setValues(int, int);
private:
    int getArea(void);
};
```

Rectangle.cpp

```
#include "Rectangle.h"

void Rectangle::setValues(int x, int y) {
     mWidth = x;
     mHeight = y;
}

int getArea() {
     return mWidth*mHeight;
}
```

main.cpp

```
#include "Rectangle.h"
#include <iostream>
using namespace std;

int main() {
     Rectangle rect;
     rect.setValue(3, 4);
     cout << "area = " << rect.getArea();
     return 0;
}
```



CLASS CONSTRUCTOR

A class *constructor* is a special member function of a class that is executed whenever we create new objects of that class.

```
class Rectangle {
private:
    int mWidth;
    int mHeight;
public:
    Rectangle(); // This is the default constructor

    void setValues(int, int);
    int getArea(void);
};

Rectangle::Rectangle() {
    cout << "Object is being created" << endl;
}
...
```

```
#include "Rectangle.h"
#include <iostream>
using namespace std;

int main() {
    Rectangle rect;
    rect.setValue(3, 4);
    cout << "area = " << rect.getArea();
    return 0;
}
```

PARAMETERIZED CLASS CONSTRUCTOR

A default constructor does not have any parameter, but if you need, a constructor can have parameters

```
class Rectangle {
private:
    int mWidth;
    int mHeight;
public:
    Rectangle(); // This is the constructor
    Rectangle(int width, int height);
    void setValues(int, int);
    int getArea(void);
};

...
Rectangle::Rectangle(int width, int height) {
    mWidth = width;
    mHeight = height;
}...
```

```
#include "Rectangle.h"
#include <iostream>
using namespace std;

int main() {
    Rectangle rect(3, 4);
    cout << "area = " << rect.getArea();
    return 0;
}
```

CLASS DESTRUCTOR

A *destructor* is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.

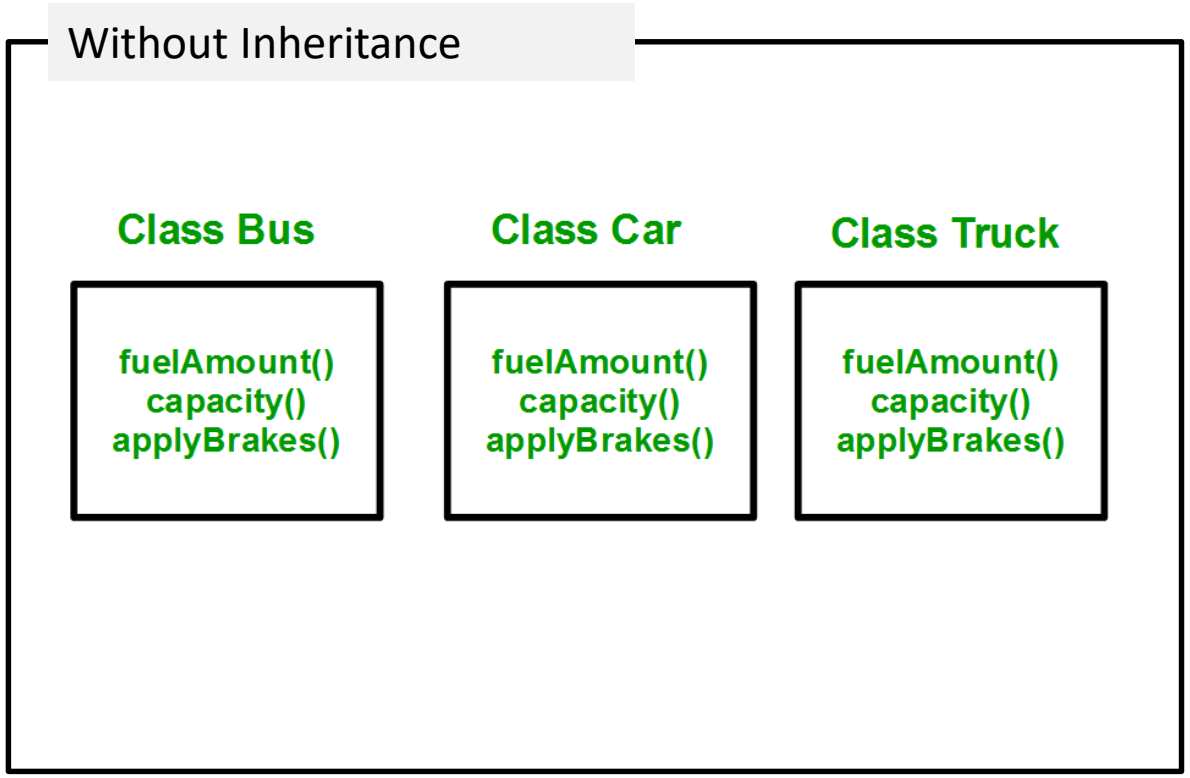
```
class Rectangle {
private:
    int mWidth;
    int mHeight;
public:
    Rectangle(); // This is the default constructor
    ~Rectangle(); // This is the destructor
    void setValues(int, int);
    int getArea(void);
};
...
Rectangle::~Rectangle() {
    cout << "Object is being deleted" << endl;
}
...
```

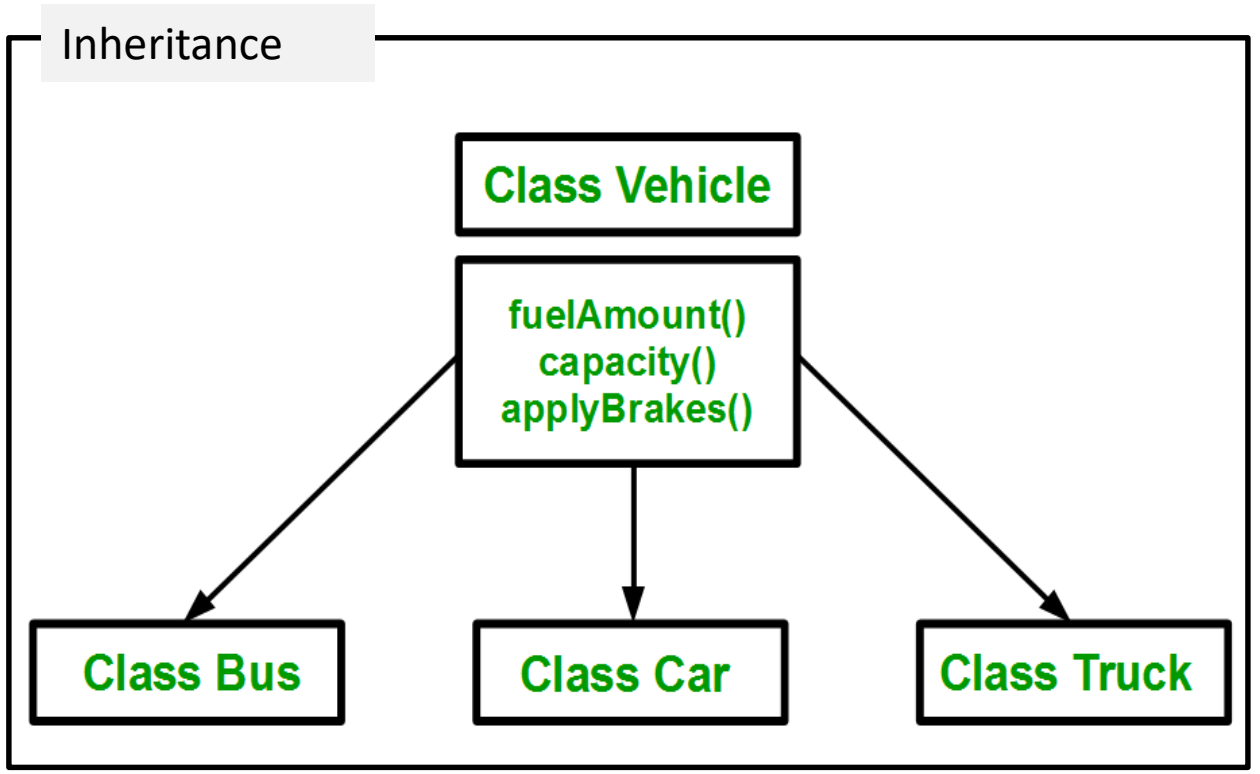
```
#include "Rectangle.h"
#include <iostream>
using namespace std;

int main() {
    Rectangle *rect = new Rectange();

    if (rect != NULL) {
        rect->setValue(3, 4);
        cout << "area = " << rect->getArea();
        delete rect;
    }


    return 0;
}
```

Inheritance (1)

The capability of a class to derive properties and characteristics from another class is called **Inheritance.**

Why and when to use inheritance?





Inheritance (2)

- **Implementing inheritance in C++**

Syntax:

```
class subclass_name : access_mode base_class_name
{
    //body of subclass
};
```

Example:

```
class Vehicle{
private:
     int mFuelAmount;
public:
     Vehicle( int fuel);
     void fuelAmount();
};

...
Vehicle :: Vehicle(int fuel) {
      mFuelAmount = fuel;
}

Vehicle :: fuelAmount() {
     std::cout << “Fuel amount” << mFuelAmount;
}
```

```
class Car : public Vehicle
{
     Car(int fuel);
};
Car:: Car(int
fuel):Vehicle(fuel){}
class Bus: public Vehicle
{

};
... → Giống Car
class Truck: public Vehicle
{

};
... → Giống Car
```

```
#include “Vehicle.h"

int main() {

      Car lexus (500);
      Bus transeco (800);
      Truck hino (1000);

      lexus.fuelAmount();
      transeco.fuelAmount();
      hino.fuelAmount();



     return 0;
}
```

Inheritance (3)

- **What will be inherited**

A derived class inherits all base class methods with the following exceptions:

- Constructors, destructors and copy constructors of the base class.
- Overloaded operators of the base class(*) .
- The friend functions of the base class.

Modes of Inheritance

	Derived Class	Derived Class	Derived Class
Base class	**Public Mode**	**Private Mode**	**Protected Mode**
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

Inheritance (4)

- **Multiple Inheritance**

Syntax

class derived-class: access mode baseA, access mode baseB....

```
// Base class Shape
class Shape {
  public:
    void setWidth(int w) {
      width = w;
    }
    void setHeight(int h) {
      height = h;
    }
  protected:
    int width, height;
};
```

```
// Base class PaintCost
class PaintCost {
  public:
    int getCost(int area) {
      return area * 70;
    }
};
```

```
// Derived class
class Rectangle: public Shape,
public PaintCost {
  public:
    int getArea() {
      return (width * height);
    }
};
```

```
int main(void) {
  Rectangle Rect;
  int area;

  Rect.setWidth(5);
  Rect.setHeight(7);

  area = Rect.getArea();
cout << "Total area: " <<
Rect.getArea() << endl;

cout << "Total paint cost: $" <<
Rect.getCost(area) << endl;

  return 0;
}
```

Overloading

C++ allows you to specify more than one definition for a **function** name or an **operator** in the same scope, which is called **function overloading** and **operator overloading(*)** respectively

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

class printData {
  public:
    void print(int i) {
      cout << "Printing int: " << i << endl;
    }
    void print(double  f) {
      cout << "Printing float: " << f << endl;
    }
    void print(char* c) {
      cout << "Printing character: " << c << endl;
    }
};
```

```
int main(void) {
   printData pd;

   // Call print to print integer
   pd.print(5);

   // Call print to print float
   pd.print(500.263);

   // Call print to print character
   pd.print("Hello C++");

   return 0;
}
```

Polymorphism (Override)

- C++ polymorphism means that a call to a member function will cause a different function to be executed **depending on the type of object** that invokes the function.

```
class Base
{
  public:
  void show()
  {
    cout << "Base class\n";
  }
};
```

```
class Derived:public Base
{
   public:
   void show()
  {
      cout << "Derived Class\n";
  }
}
```

```
int main()
{
  Base b;       //Base class object
  Derived d;    //Derived class object
  b.show();
  d.show();
}
```

VIRTUAL FUNCTION

- A **virtual** function is a function in a base class that is declared using the keyword **virtual**.

```
class Base
{
  public:
  virtual void show()
  {
    cout << "Base class\n";
  }
};
```

```
class Derived:public Base
{
   public:
   void show()
  {
      cout << "Derived Class\n";
  }
}
```

```
int main()
{
  Base *b;      //Base class object
  Derived d;    //Derived class object
  b = &d;
  b->show();
}
```

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. his sort of operation is referred to as **dynamic linkage**, or **late binding**.

Note: Without virtual → **Static linkage** or **early binding**

Abstract Class AND Pure Virtual Functions

- **Pure Virtual Functions**

Pure virtual Functions are virtual functions with no definition. They start with **virtual** keyword and ends with = 0. Here is the syntax for a pure virtual function

Example:

```
virtual void f() = 0;
```

- **Abstract Class**

Abstract Class is a class which contains at least one Pure Virtual function in it

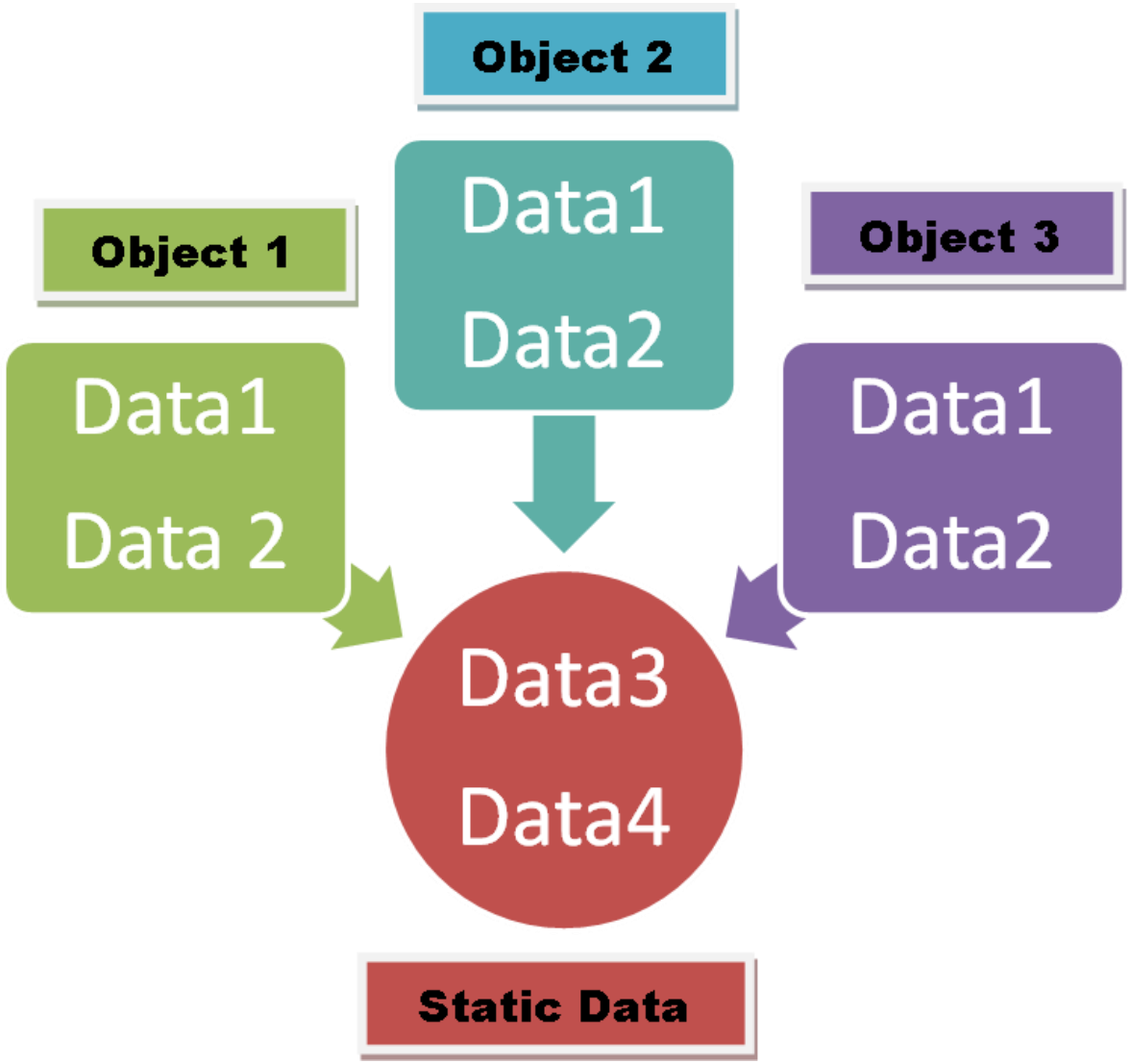
```
class Base    //Abstract base class
{
 public:
 //Pure Virtual Function
 virtual void show() = 0;
};
```

```
class Derived:public Base
{
 public:
 void show()
 { cout << "Implementation of Virtual
Function in Derived class";
 }
};
```

```
int main()
{
 Base obj;      //Compile Time Error
 Base *b;
 Derived d;
 b = &d;
 b->show();
}
```

STATIC DATA OF A C++ CLASS (1)

- A static data exists throughout the whole life of the program.
- A static data is shared by all objects of the class



STATIC DATA OF A C++ CLASS (2)

```
#include <iostream>
using namespace std;
class Class {
public:
    static int mStaticData;
    int mNonStaticData;
    void print(void);
};
int Class::mStaticData = 0; // definition & initializer
void Class::print(void) {
    cout << "Static = " << ++mStaticData << endl;
    cout << "NonStatic = " << mNonStaticData << endl;
}
```

```
int main(void) {
    Class object1, object2;
    object1.mNonStaticData = 10;
    object2.mNonStaticData = 20;
    object1.print();
    object2.print();
    cout << "Static = " <<
Class::mStaticData << endl;
    return 0;
}
```

Result

```
Static = 1
NonStatic = 10
Static = 2
NonStatic = 20
Static = 2
```

STATIC FUNCTIONS OF A C++ CLASS (1)

- In class, functions can also be declared as static
- Some properties of static functions are:
  - Can access only othes static members (data or functions)
  - Can be called using class name without object

  class_name::function_name

  - Have no this pointer

STATIC FUNCTIONS OF A C++ CLASS (2)

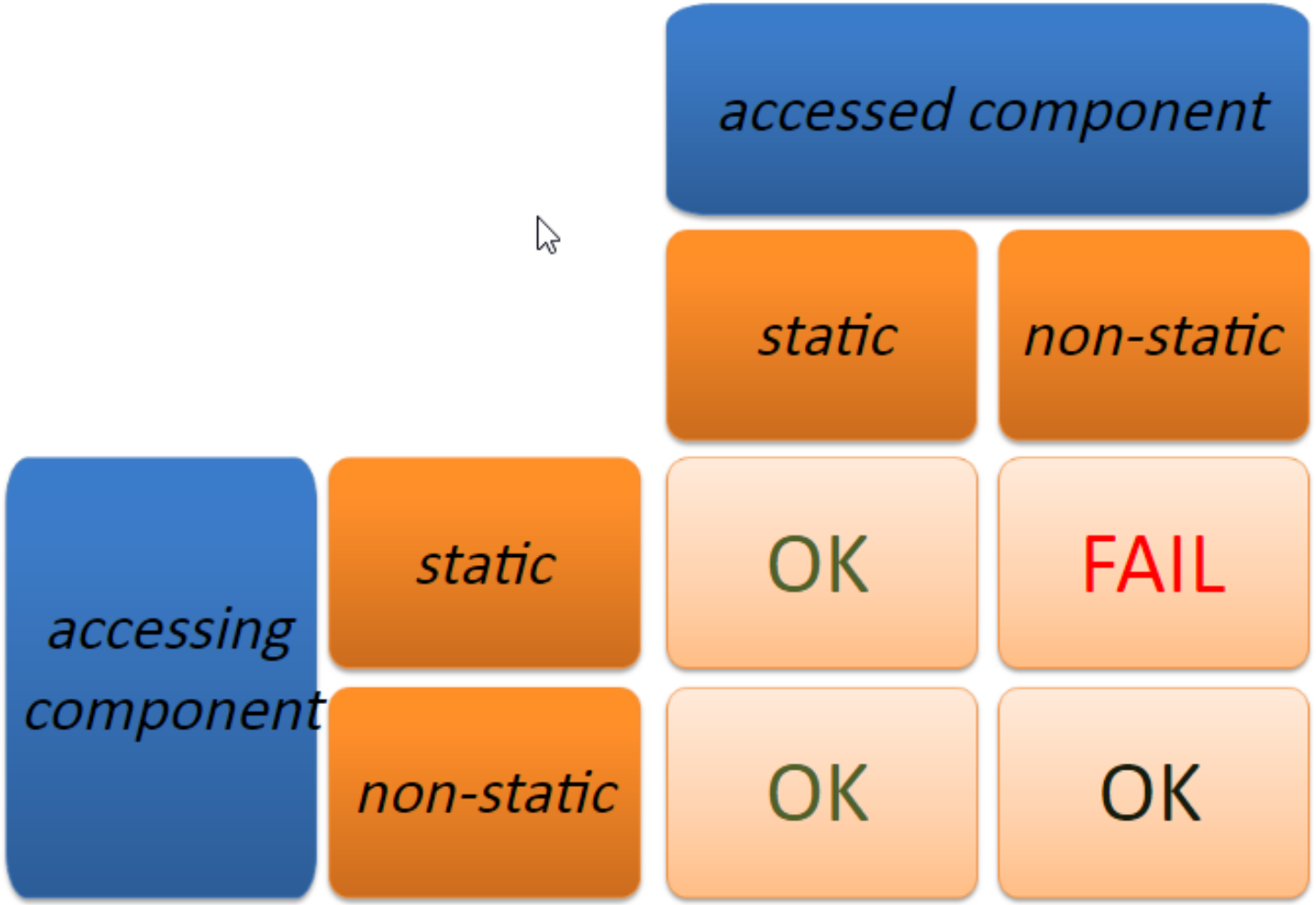
```
#include <iostream>
using namespace std;
class Class {
public:
    static int mStaticData;
    int mNonStaticData;
    void print(void);
    static void staticFunction();
};
...
void Class::staticFunction() {
    cout << "Static = " << mStaticData << endl;
}
```

```
int main(void) {
    Class instance1, instance2;
    instance1.mNonStaticData = 10;
    instance2.mNonStaticData = 20;
    instance1.print();
    instance2.print();
    instance1.staticFunction();
    instance2.staticFunction();
    Class::staticFunction();
    return 0;
}
```

Result

```
Static = 1
NonStatic = 10
Static = 2
NonStatic = 20
Static = 2
Static = 2
Static = 2
```

STATIC MEMBERS ACCESS TABLE



FRIEND CLASS

```
class Node {
private:
    int key;
    Node* next;
    /* Other members of Node Class */

    // Now class  LinkedList can
    // access private members of Node
    friend class LinkedList;
};
```

Friend Class A friend class can access private and protected members of other class in which it is declared as friend. It is sometimes useful to allow a particular class to access private members of other class. For example a LinkedList class may be allowed to access private members of Node.

FRIEND FUNCTION

```
class Node {
private:
    int key;
    Node* next;
    /* Other members of Node Class */

    friend int LinkedList::search();
    // Only search() of linkedList
    // can access internal members
};
```

Like friend class, a friend function can be given special grant to access private and protected members. A friend function can be:

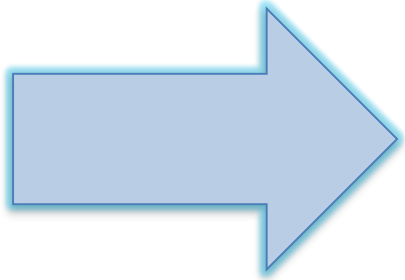
a) A method of another class

b) A global function

FRIEND CLASS AND FUNCTION (1)

```
#include <iostream>
class B;
class A {
public:
   void showB(B&);
};
class B {
private:
   int b;
public:
   B() { b = 0; }
   friend void A::showB(B& x); // Friend function
};
void A::showB(B& x)
{
   // Since showB() is friend of B, it can
   // access private members of B
   std::cout << "B::b = " << x.b;
}
int main()
{
   A a;
   B x;
   a.showB(x);
   return 0;
}
```

A simple and complete C++ program to demonstrate friend Class



Output:

```
A::a=0
```

FRIEND CLASS AND FUNCTION (2)

```
#include <iostream>
class A {
private:
   int a;
public:
   A() { a = 0; }
   friend class B; // Friend Class
};
class B {
private:
   int b;
public:
   void showA(A& x)
   {
      // Since B is friend of A, it can access
      // private members of A
      std::cout << "A::a=" << x.a;
   }
};
int main()
{
   A a;
   B b;
   b.showA(a);
   return 0;
}
```

A simple and complete C++ program to demonstrate friend function of another class

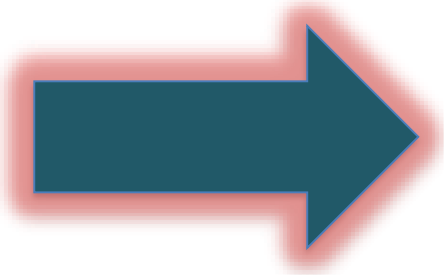
Output:

```
A::a=0
```

FRIEND CLASS AND FUNCTION (3)

```
#include <iostream>
class A {
    int a;
public:
    A() { a = 0; }
    // global friend function
    friend void showA(A&);
};
void showA(A& x)
{
    // Since showA() is a friend, it can access
    // private members of A
    std::cout << "A::a=" << x.a;
}
int main()
{
    A a;
    showA(a);
    return 0;
}
```

A simple and complete C++ program to demonstrate friend function of another class



Output:

```
A::a=0
```