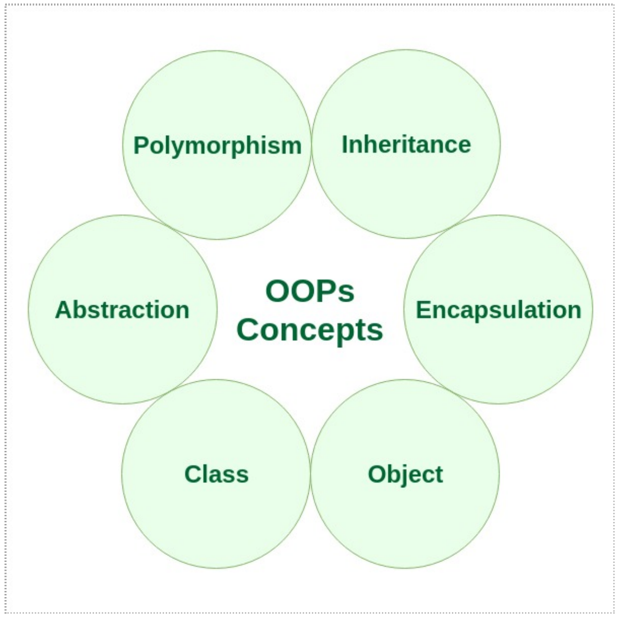
Data Structures and Algorithms

OOP concepts

Object Oriented Programming - OOP

- The major purpose of C++ programming is to introduce the concept of object orientation to the C programming language.
- Object Oriented Programming is a paradigm that provides many concepts such as inheritance, data binding, polymorphism etc.
- Object: Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.
- Class: Collection of objects is called class. It is a logical entity.
- Inheritance: When one object acquires all the properties and behaviors edinmek/kazanmak of parent object i.e. known as inheritance. It provides code reusability. It tekrar kullanılabilirlik is used to achieve runtime polymorphism.
- Polymorphism: When one task is performed by different ways i.e. known as polymorphism. For example: to convince the customer differently, to draw something e.g. shape or rectangle etc. In C++, we use Function overloading and Function overriding to achieve polymorphism.
- Abstraction: Hiding internal details and showing functionality is known as abstraction. For example: phone call, we don't know the internal processing. In C++, we use abstract class and interface to achieve abstraction.
- Encapsulation: Binding (or wrapping) code and data together into a single unit is known as encapsulation. For example: capsule, it is wrapped with different medicines.



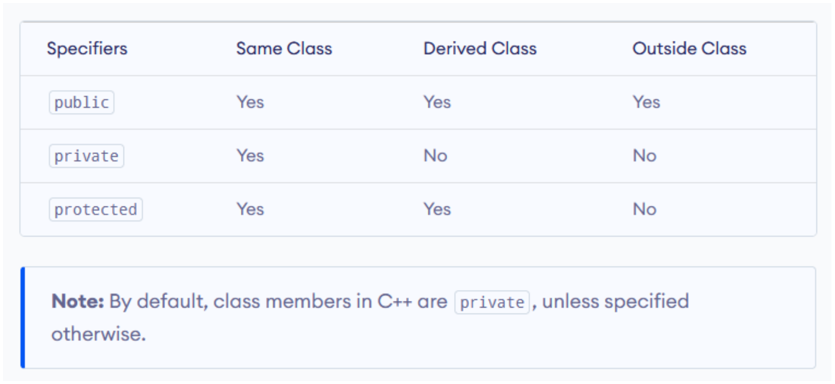
Access Modifiers

- One of the main features of object-oriented programming languages such as C++ is **data hiding**.
- Data hiding refers to restricting access to data members of a class. This is to prevent other functions and classes from tampering with the class data.

kısıtlama

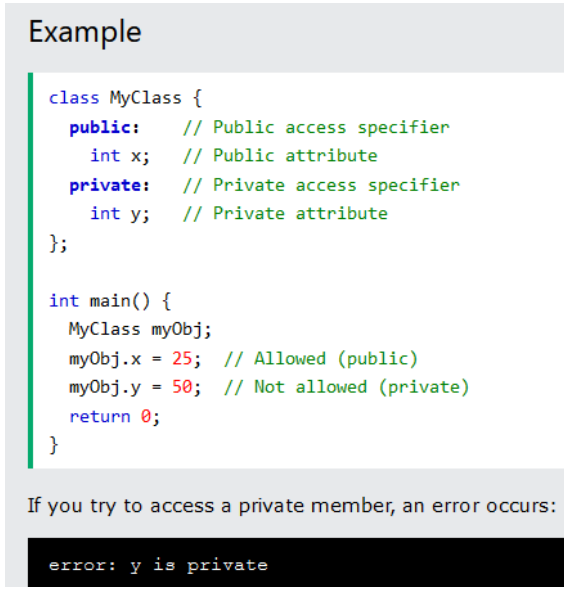
kurcalama/karışma

- However, it is also important to make some member functions and member data accessible so that the hidden data can be manipulated indirectly.
- The access modifiers of C++ allows us to determine which class members are accessible to other classes and functions, and which are not.
- A **public** member is accessible from anywhere outside the class but within a program. You can set and get the value of public variables without any member function
- A **private** member variable or function cannot be accessed, or even viewed from outside the class. Only the class and friend functions can access private members.
- A **protected** member variable or function is very similar to a private member but it provided one additional benefit that they can be accessed in child classes which are called derived classes.



Specifiers	Same Class	Derived Class	Outside Class
`public`	Yes	Yes	Yes
`private`	Yes	No	No
`protected`	Yes	Yes	No

Note: By default, class members in C++ are `private`, unless specified otherwise.



Example

```
class MyClass {
  public:    // Public access specifier
    int x;   // Public attribute
  private:   // Private access specifier
    int y;   // Private attribute
};

int main() {
  MyClass myObj;
  myObj.x = 25;  // Allowed (public)
  myObj.y = 50;  // Not allowed (private)
  return 0;
}
```

If you try to access a private member, an error occurs:

```
error: y is private
```

https://www.programiz.com/cpp-programming/access-modifiers
https://www.tutorialspoint.com/cplusplus/cpp_class_access_modifiers.htm

Class definition

- There are two ways to define functions that belongs to a class:

Inside class definition

```
class MyClass {   // The class
  public:         // Access specifier

    // Method/function defined inside the
class
    void myMethod() {
     cout << "Hello World!";
    }
};

int main() {
   // Create an object of MyClass
  MyClass myObj;
  // Call the method
  myObj.myMethod();
 return 0;
}
```

Outside class definition

```
class MyClass {         // The class
  public:               // Access specifier
    void myMethod();    // Method/function
declaration
};

// Method/function definition outside the
class
void MyClass::myMethod() {
  cout << "Hello World!";
}

int main() {
  // Create an object of MyClass
  MyClass myObj;
  // Call the method
  myObj.myMethod();
return 0;
}
```

Class definition

- Example:

```cpp
#include <iostream>

using namespace std;

class Box {
   public:
      double length;         // Length of a box
      double breadth;        // Breadth of a box
      double height;         // Height of a box

      // Member functions declaration
      double getVolume(void);
      void setLength( double len );
      void setBreadth( double bre );
      void setHeight( double hei );
};

// Member functions definitions
double Box::getVolume(void) {
   return length * breadth * height;
}

void Box::setLength( double len ) {
   length = len;
}
void Box::setBreadth( double bre ) {
   breadth = bre;
}
void Box::setHeight( double hei ) {
   height = hei;
}
```

we define the parameters of the function then outside from the class we call the methods and we pair the parameters

creating an object

```cpp
// Main function for the program
int main() {
   Box Box1;              // Declare Box1 of type Box
   Box Box2;              // Declare Box2 of type Box
   double volume = 0.0; // Store the volume of a box
here

   // box 1 specification
   Box1.setLength(6.0);
   Box1.setBreadth(7.0);
   Box1.setHeight(5.0);

   // box 2 specification
   Box2.setLength(12.0);
   Box2.setBreadth(13.0);
   Box2.setHeight(10.0);

   // volume of box 1
   volume = Box1.getVolume();
   cout << "Volume of Box1 : " << volume <<endl;

   // volume of box 2
   volume = Box2.getVolume();
   cout << "Volume of Box2 : " << volume <<endl;
   return 0;
}
```

Constructors

- A constructor in C++ is a special method that is automatically called when an object of a class is created.
- To create a constructor, use **the same name as the class**, followed by parentheses ()

```
class Car {         // The class
  public:           // Access specifier
    string brand;   // Attribute
    string model;   // Attribute
    int year;       // Attribute
    Car(string x, string y, int z) { // Constructor with parameters
      brand = x;
      model = y;
      year = z;
    }
};

int main() {
  // Create Car objects and call the constructor with different values
  Car carObj1("BMW", "X5", 1999);
  Car carObj2("Ford", "Mustang", 1969);

  // Print values
  cout << carObj1.brand << " " << carObj1.model << " " << carObj1.year << "\n";
  cout << carObj2.brand << " " << carObj2.model << " " << carObj2.year << "\n";
  return 0;
```

Constructors

- A copy constructor is a member function that initializes an object using another object of the same class.
- A copy constructor has the following general function prototype:

```
ClassName (const ClassName &copied_obj);
```

https://www.geeksforgeeks.org/copy-constructor-in-cpp/

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

class Point
{
private:
    int x, y;
public:
    Point(int x1, int y1) { x = x1; y = y1; }

    // Copy constructor
    Point(const Point &p1) {x = p1.x; y = p1.y; }

    int getX()            {  return x; }
    int getY()            {  return y; }
};

int main()
{
    Point p1(10, 15); // Normal constructor is called here
    Point p2 = p1; // Copy constructor is called here

    // Let us access values assigned by constructors
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
    cout << "\np2.x = " << p2.getX() << ", p2.y = " <<
p2.getY();

    return 0;
}
```

Destructors

- Destructor is an instance member function which is invoked automatically çağırıldı whenever an object is going to be destroyed. Meaning, a destructor is the last function that is going to be called before an object is destroyed.
- The thing is to be noted here, if the object is created by using new or the constructor uses new to allocate tahsis etme/ayırmak memory which resides in the heap memory or the free store, the destructor should use delete to free the memory.

```
class String {
private:
    char* s;
    int size;

public:
    String(char*); // constructor
    ~String(); // destructor
};

String::String(char* c)
{
    size = strlen(c);
    s = new char[size + 1];
    strcpy(s, c);
}
String::~String() {
 delete[] s;
}
```

Encapsulation

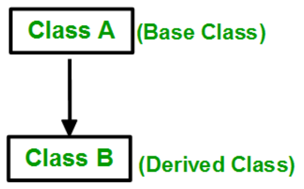
- The meaning of Encapsulation, is to make sure that "sensitive" data is hidden from users.
- To achieve this, you must declare class variables/attributes as private (cannot be accessed from outside the class).
- If you want others to read or modify the value of a private member, you can provide public get and set method
- To access a private attribute, use public "get" and "set" methods:
- **Why Encapsulation?**
- It is considered good practice to declare your class attributes as private (as often as you can). Encapsulation ensures better control of your data, because you (or others) can change one part of the code without affecting other parts
- Increased security of data

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

class Employee {
  private:
    // Private attribute
    int salary;

  public:
    // Setter
    void setSalary(int s) {
      salary = s;
    }
    // Getter
    int getSalary() {
      return salary;
    }
};

int main() {
  Employee myObj;
  myObj.setSalary(50000);
  cout <<
myObj.getSalary();
  return 0;
}
```

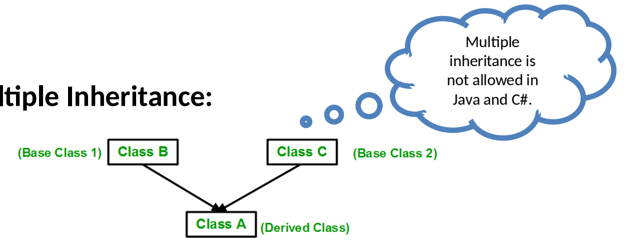
Inheritance

- The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important feature of Object Oriented Programming.
- Sub Class: The class that inherits properties from another class is called Sub class or Derived Class.
- Super Class: The class whose properties are inherited by sub class is called Base Class or Super class.

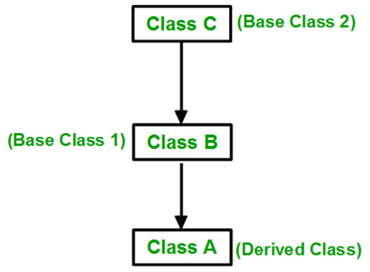
Single Inheritance:

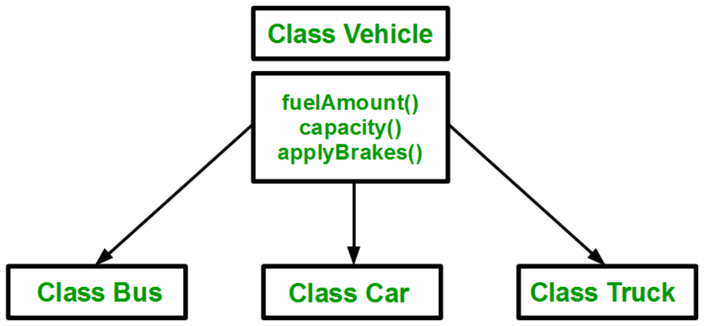


Multiple Inheritance:



Multilevel Inheritance:





https://www.geeksforgeeks.org/inheritance-in-c/

Inheritance

```
// C++ program to explain
// Single inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle {
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

// sub class derived from a single base classes
class Car: public Vehicle{

};

// main function
int main()
{
    // creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return 0;
}
// main function
int main()
{
    //creating object of sub class will
    //invoke the constructor of base classes
    Car obj;
    return 0;
}
```

```
// C++ program to explain mnultilevel inheritance

#include <iostream>
using namespace std;

// base class
class Vehicle
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

// first sub_class derived from class vehicle
class fourWheeler: public Vehicle
{  public:
    fourWheeler()
    {
      cout<<"Objects with 4 wheels are vehicles"<<endl;
    }
};
// sub class derived from the derived base class fourWheeler
class Car: public fourWheeler{
   public:
     Car()
     {
       cout<<"Car has 4 Wheels"<<endl;
     }
};

// main function
int main()
{
    //creating object of sub class will
    //invoke the constructor of base classes
    Car obj;
    return 0;
}
```

Friend class

- **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.

- A non-member function can access the private and protected members of a class if it is declared a friend of that class. That is done by including a declaration of this external function within the class, and preceding it with the keyword friend:

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

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

class Node {
private:
    int key;
    Node* next;

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

https://www.geeksforgeeks.org/friend-class-function-cpp/

Files

```cpp
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main () {
  // Create a text file
  ofstream MyWriteFile("filename.txt");

  // Write to the file
  MyWriteFile << "Files can be tricky, but it is fun enough!";

  // Close the file
  MyWriteFile.close();

  // Create a text string, which is used to output the text file
  string myText;

  // Read from the text file
  ifstream MyReadFile("filename.txt");

  // Use a while loop together with the getline() function to read
the file line by line
  while (getline (MyReadFile, myText)) {
    // Output the text from the file
    cout << myText;
  }

  // Close the file
  MyReadFile.close();
}
```

- To create a file, use either the ofstream or fstream class, and specify the name of the file.

- To write to the file, use the insertion operator (<<).
- To read from a file, use either the ifstream or fstream class, and the name of the file.

- Note that we also use a while loop together with the getline() function (which belongs to the ifstream class) to read the file line by line, and to print the content of the file:

Exceptions

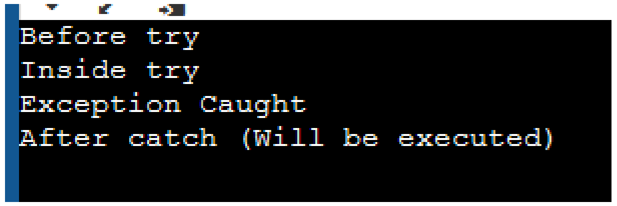
- When executing C++ code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.
- When an error occurs, C++ will normally stop and generate an error message. The technical term for this is: C++ will throw an exception (throw an error). C++ try and catch
- Exception handling in C++ consist of three keywords: try, throw and catch:
- The try statement allows you to define a block of code to be tested for errors while it is being executed.
- The throw keyword throws an exception when a problem is detected, which lets us create a custom error.
- The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.
- The try and catch keywords come in pairs

```
try {
  // Block of code to try
  throw exception; // Throw an exception when a problem arise
}
catch () {
  // Block of code to handle errors
}
```

- All the Errors are Exceptions but the reverse is not true.
- In general Errors are which nobody can control or guess when it happened, on the other hand Exception can be guessed and can be handled.
- An Error is something that most of the time you cannot handle it. Errors are unchecked exception and the developer is not required to do anything with these. Errors normally tend to signal the end of your program, it typically cannot be recovered from and should cause you exit from current program. It should not be caught or handled.

Exceptions

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

int main()
{
   int x = -1;

   // Some code
   cout << "Before try \n";
   try {
      cout << "Inside try \n";
      if (x < 0)
      {
         throw x;
         cout << "After throw (Never executed) \
n";
      }
   }
   catch (int x ) {
      cout << "Exception Caught \n";
   }

   cout << "After catch (Will be executed) \n";
   return 0;
}
```



```
Before try
Inside try
Exception Caught
After catch (Will be executed)
```

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

double division(int a, int b) {
   if( b == 0 ) {
      throw "Division by zero condition!";
   }
   return (a/b);
}

int main () {
   int x = 50;
   int y = 0;
   double z = 0;

   try {
      z = division(x, y);
      cout << z << endl;
   } catch (const char* msg) {
     cerr << msg << endl;
   }

   return 0;
}
```

y is especially made zero to produce exception

Exceptions

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

struct MyException : public exception {
   const char * what () const throw () {
      return "C++ Exception";
   }
};

int main() {
   try {
      throw MyException();
   } catch(MyException& e) {
      cout << "MyException caught" <<
endl;
      cout << e.what() << endl;
   } catch(std::exception& e) {
      //Other errors
   }
}
```

- You can define your own exceptions by inheriting and overriding exception class functionality. Following is the example, which shows how you can use std::exception class to implement your own exception in standard way
- Here, what() is a public method provided by exception class and it has been overridden by all the child exception classes. This returns the cause of an exception.