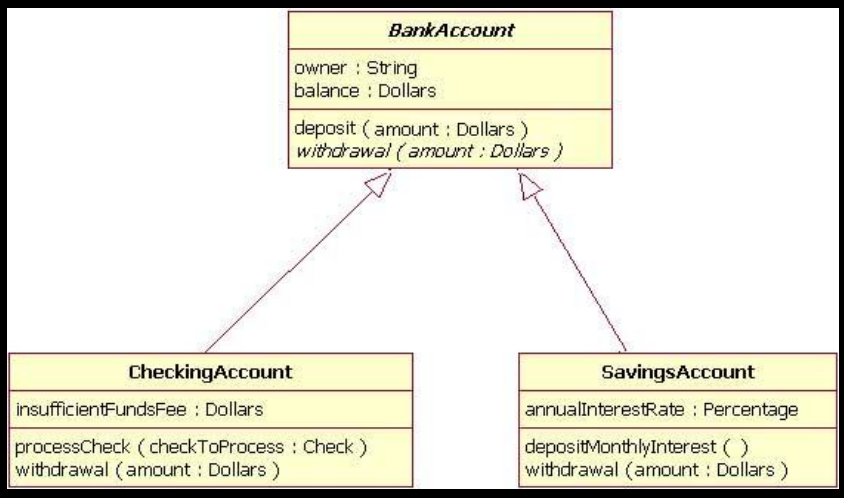
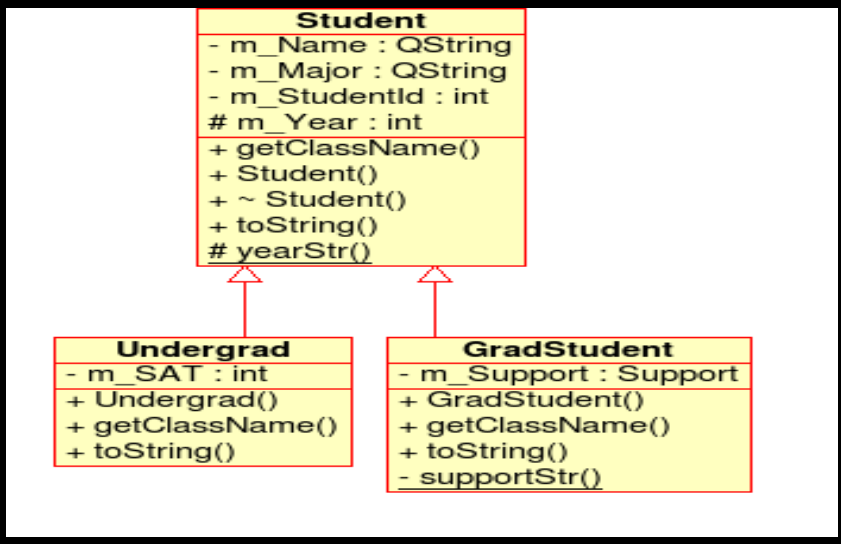
Week 08

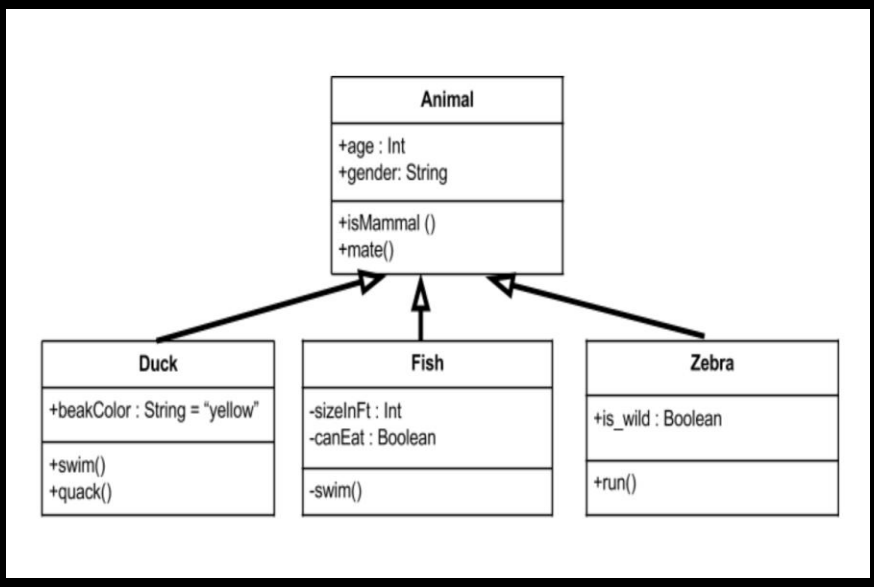
- Is a relationship (contd.)
- Multiple Inheritance
- Pointer to Objects
- Static type and Dynamic Type
- Polymorphism
- Compile Time Vs Runtime Polymorphism
- Compile Time Polymorphism (with examples)

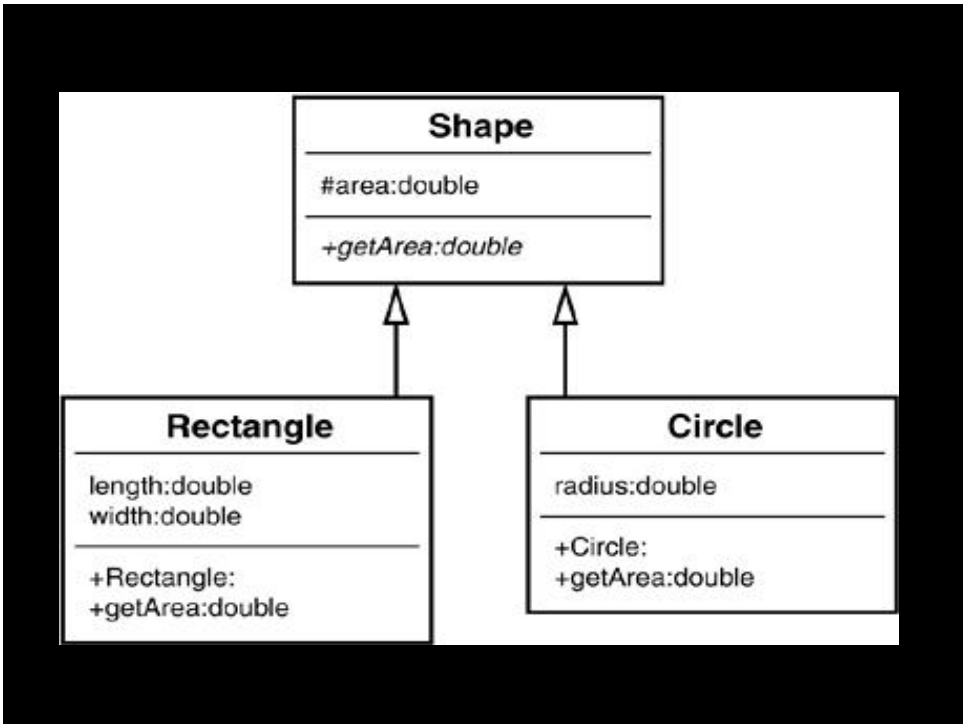
UML representation of Inheritance



UML representation of Inheritance







Your Turn

- Write a program with a mother class animal. Inside it define a name and an age variables, and set_value() function.Then create two classes Zebra and Dolphin which write a message telling the age, the name and giving some extra information (e.g. place of origin).

Multiple Inheritance

- We may want to reuse characteristics of more than one parent class

Single/Multiple Inheritance

Single Inheritance

Each class or instance object has a single parent

Multiple Inheritance

Classes inherit from multiple base classes (might not have same ancestors as shown in the example below)

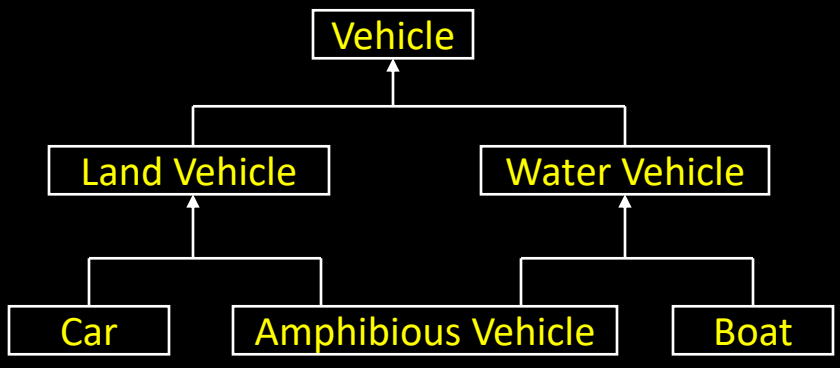
Defines a relationship

Between several (independent) class types

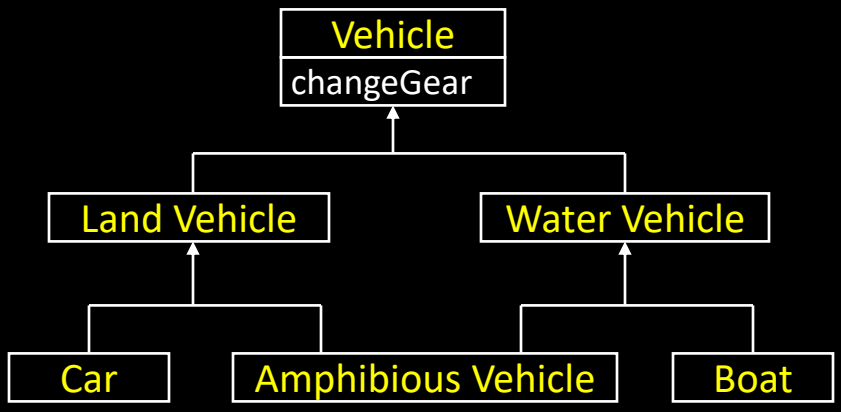
Example – Multiple Inheritance

Amphibious Vehicle

Example – Multiple Inheritance



Problem – Duplicate Features (Diamond Problem)

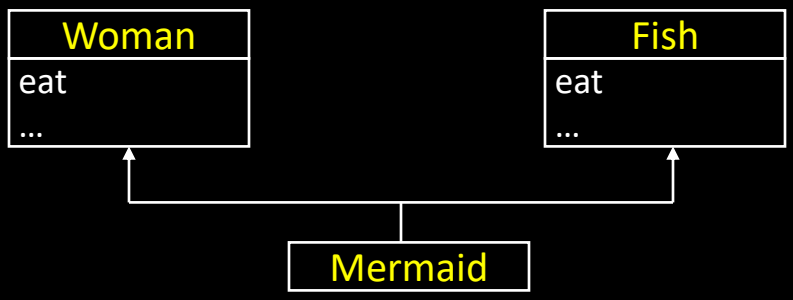


- Which *changeGear* operation Amphibious Vehicle inherits?

Example – Multiple Inheritance

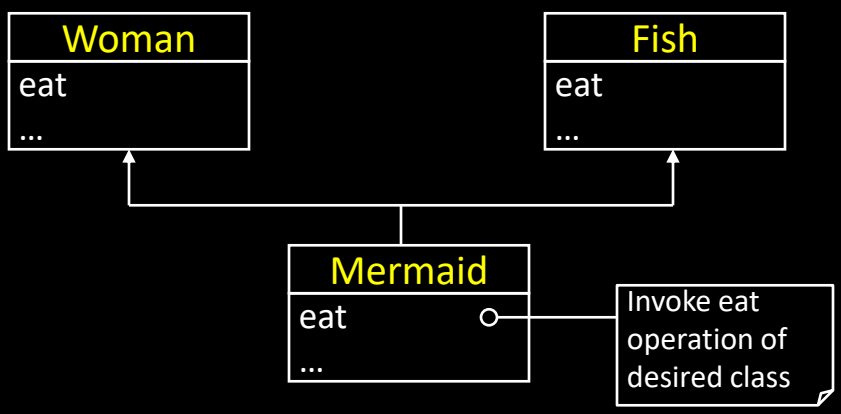
Mermaid

Problem – Duplicate Features

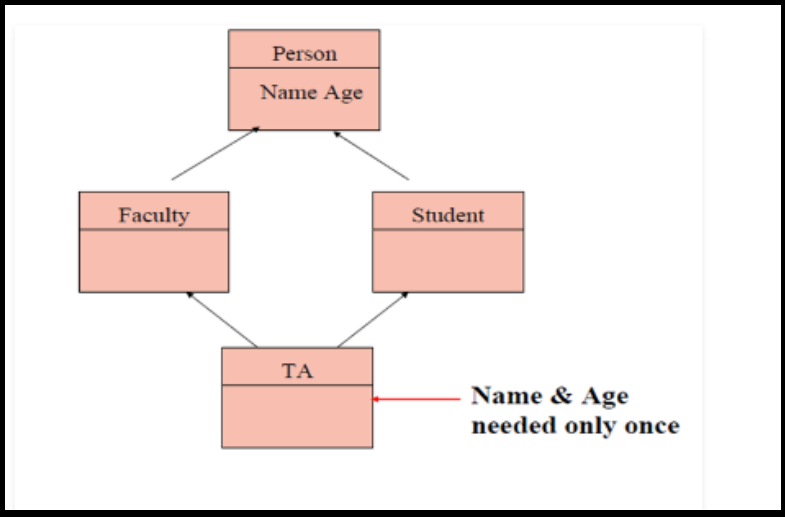


- Which *eat* operation *Mermaid* inherits?

Solution – Override the Common Feature

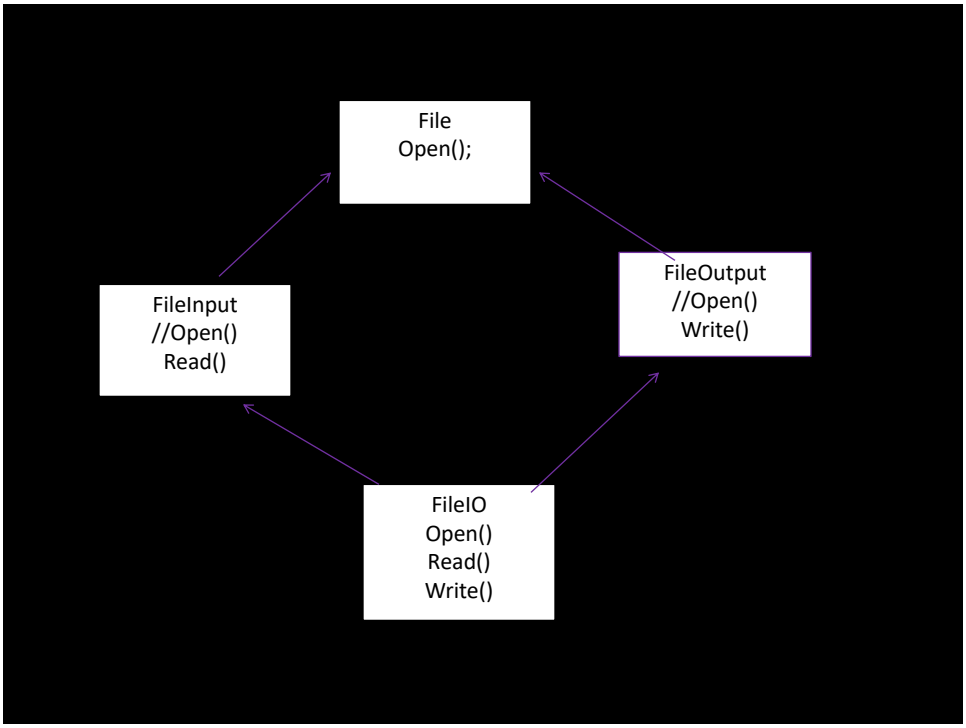


Example



Problems with Multiple Inheritance

- Increased complexity
- Reduced understanding
- Duplicate features



Solution to Diamond Problem

- Some languages disallow diamond hierarchy
- Others provide mechanism to ignore characteristics from one side
- C++ provide flexibility to use virtual inheritance.

Virtual inheritance

- **Virtual inheritance** is a C++ technique that ensures only one copy of a base class's member variables are **inherited** by grandchild derived classes.
- In C++, a base class intended to be common throughout the hierarchy is denoted as **virtual** with the **virtual** keyword

Multiple Inheritance Code Examples on Compiler

Pointer to Objects

- Pointer to objects are similar as pointer to built-in types
- They can also be used to dynamically allocate objects

Example

```
class Student{
…
public:
  Studen();
  Student(char * aName);
  void setRollNo(int aNo);
};
```

Example

```
int main(){
  Student obj;
  Student *ptr;
  ptr = &obj;
  ptr->setRollNo(10);
  return 0;
}
```

Allocation with new Operator

- new operator can be used to create objects at runtime

Example

```
int main(){
  Student *ptr;
  ptr = new Student;
  ptr->setRollNo(10);
  return 0;
}
```

Example

```
int main(){
  Student *ptr;
  ptr = new Student("Ali");
  ptr->setRollNo(10);
  return 0;
}
```

Example

```
int main()
{
  Student *ptr = new Student[100];
  for(int i = 0; i < 100;i++)
  {
     ptr->setRollNo(10);
  }
  return 0;
}
```

Breakup of new Operation

- new operator is decomposed as follows
  - Allocating space in memory
  - Calling the appropriate constructor

"IS A" Relationship

- Public inheritance models the "IS A" relationship
- Derived object IS A kind of base object

Example

```
class Person {
   char * name;
public: ...
   const char * GetName();
};
class Student: public Person{
   int rollNo;
public: ...
   int GetRollNo();
};
```

Example

```
int main()
{
   Student sobj;
   cout << sobj.GetName();
   cout << sobj.GetRollNo();
   return 0;
}
```

“IS A” Relationship

- The base class pointer can point towards an object of derived class

Example

```
int main(){
 Person * pPtr = 0;
 Student s;
 pPtr = &s;
 cout << pPtr->GetName();

 return 0;
}
```

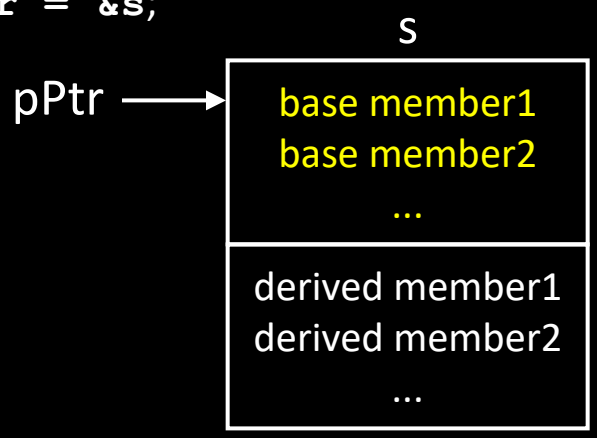
Example

```
int main(){
 Person * pPtr = 0;
 Student s;
 pPtr = &s;
 cout << pPtr->GetName();

 return 0;
}
```

Example

```
pPtr = &s;
```



Example

```
int main(){
 Person * pPtr = 0;
 Student s;
 pPtr = &s;
 //Error
 cout << pPtr->GetRollNo();
 return 0;
}
```

Static Type

- The type that is used to declare a reference or pointer is called its static type
  - The static type of pPtr is Person
  - The static type of s is Student

Member Access

- The access to members is determined by static type
- The static type of pPtr is Person

- Following call is erroneous

```
pPtr->GetRollNo();
```

“IS A” Relationship

- We can use a reference of derived object where the reference of base object is required

Example

```
int main(){
   Person p;
   Student s;
   Person & refp = s;
   cout << refp.GetName();
   cout << refp.GetRollNo(); //Error
   return 0;
}
```

Example

```
void Play(const Person& p){
   cout << p.GetName()
        << " is playing";
}
void Study(const Student& s){
   cout << s.GetRollNo()
        << " is Studying";
}
```

Example

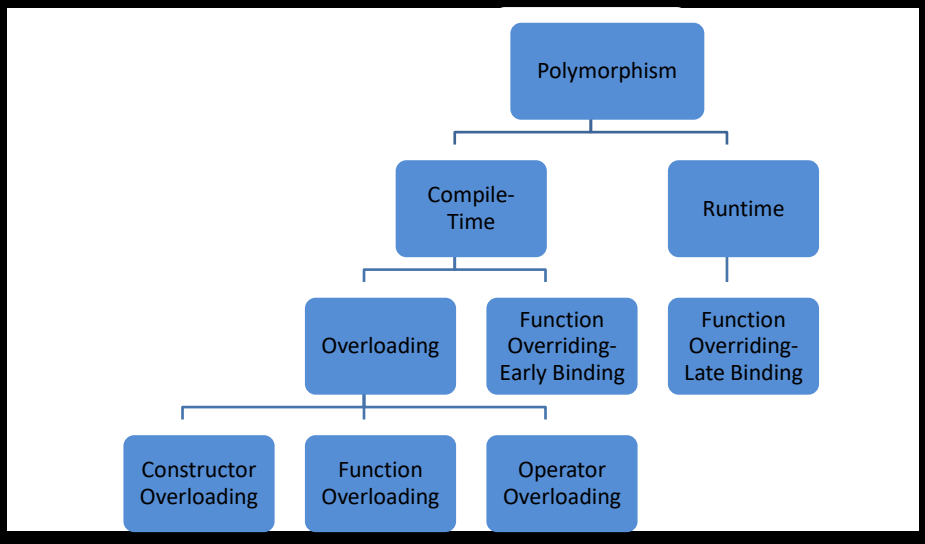
```
int main(){
   Person p;
   Student s;
   Play(p);
   Play(s);
   return 0;
}
```

Polymorphism

- Polymorphism can be defined as the ability of a message to be displayed in more than one form.
- For example,
  - a person at a same time can have different characteristic. Like a man at a same time is a father, a husband, a employee. So a same person have different behavior in different situations. This is called polymorphism.

Polymorphism

- Poly-Many, Morphs-Forms



Compile Time Polymorphism

- Overloading
  - Constructor Overloading (already discussed)
  - Function Overloading
  - Operator Overloading (will discuss later)
- Overriding (Early Binding)

Function Overloading

- Function overloading is a feature in C++ where two or more functions can have the same name but different parameters.
- Two overloaded **functions** must not have the same **signature**. The return value is not part of a **function's signature**.

Function Signature

- A **function's signature** includes the **function's** name and the number, order and type of its formal parameters. Two overloaded **functions** must not have the same **signature**. The return value is not part of a **function's signature**.

Function Overloading

```
void print(int i) {
  cout << " Here is int " << i << endl;
}
void print(double  f) {
  cout << " Here is float " << f << endl;
}
void print(char* c) {
  cout << " Here is char* " << c << endl;
}
```

Function Overloading

```
int main() {
  print(10);
  print(10.10);
  print("ten");
  return 0;
}
```

Functions that cannot be overloaded in C++

Function declarations that differ only in the return type. For example, the following program fails in compilation.

```
int foo() {
  return 10;
}
char foo() {
  return 'a';
}
```

```
int main()
{
   char x = foo();
   getchar();
   return 0;
}
```

Function Overriding

- It is the redefinition of base class function in its derived class with same signature return type and parameters.
- It can only be done in derived class.

Function Overriding

- A class may need to override the default behaviour provided by its base class
- Reasons for overriding
  - Provide behaviour specific to a derived class
  - Extend the default behaviour
  - Restrict the default behaviour
  - Improve performance

Function Overriding (Early Binding)

```
class A{
public:
void print(){
cout<<"I am from A";
}
};
class B:public A{
public:
void print(){
cout <<"I am from B";
}
};
```

```
int main(){
A objA;
B objB;
A* aptr;
objA.print(); // I am from A
objB.print(); // I am from B
aptr = &objA;
aptr -> print(); //I am from A
aptr = &objB;
aptr->print(); //I am from A (due
   to early binding, compiler will
   bind this function call to the
   static type of aptr i.e. class A
}
```

Function Overloading VS Function Overriding

1. Overriding of functions occurs when one class is inherited from another class.
2. Overloaded functions must differ in function signature i.e. either number of parameters or type of parameters should differ.
3. Overridden functions are in different scopes

1. Overloading can occur without inheritance.
2. In overriding, function signatures must be same.
3. Overloaded functions are in same scope.

Review