A C++ INTRODUCTION

C++ Objects and Classes (1/2)

- A class is a user-defined, abstract data type
- An object is a class instance
- Classes are a way to formalize
  - Structure -> attributes -> member data
  - Behavior -> methods -> member functions
  - Hierarchical class dependencies
    - Inheritance, ‘is-a’
    - Composition, ‘contains-a’
    - Utilization, ‘use-a’
- Classes are used to
  - Encapsulate data
  - Define access interfaces

C++ Objects and Classes (2/2)

- Access to member data/functions is controlled
  - Private (default), protected, public

Static Declarations in Classes (1/2)

- Each time a new object is created, memory space for every attribute is reserved
- If key-word `static` is used with an attribute, no matter the number of instances created, memory space is reserved once.
- Static methods refer to the whole class
  - pointer `this` is not available
- Static methods can be called in two different ways
  - Using an object
  - Using the operator ::

Static Declarations in Classes (2/2)

- Example

```
// test.h
class test{
  static int x;
 public:
  test();
   static int f(void);
   // other declarations ...
};
// test.cpp
int test::x=1; // initialization
int test::f(){return x;} // ...
// main.cpp
test A;
A.f();
test::f();
```

Operators (1/1)

- Example

```
// test.h
class test{
 int x;
 public:
  test();
   test operator +(const test&);
  // other declarations ...
};
//main.cpp
test A,B,C;
C=A+B; // C=A.operator+(B);
```

Class Templates (1/2)

- Generic classes based on formal arguments
  - Data type = parameter
- Define families of classes
- Become concrete code once their definitions are applied to real entities

Class Templates (2/2)

- Example

```
template <class T>
class stack{
   public:
     stack (int c);
     ~stack();
     void push (T elem);
     T pop(void);
     bool full(void);
     bool empty(void);
   private:
   // ...
};
```

```
void main(){
  stack<int> a(10);
  int i=1;
  while(!a.full()){
       a.push(i++);
  }
  while(!a.empty()){
       cout << "\t"
            << a.pop();
  }
}
```

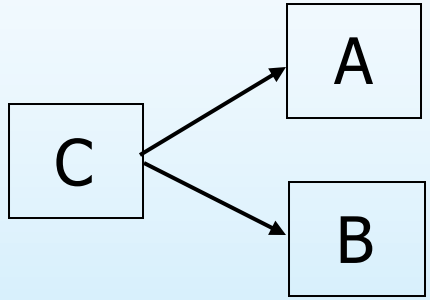
- Result
  - 10 9 8 7 6 5 4 3 2 1

C++ Inheritance

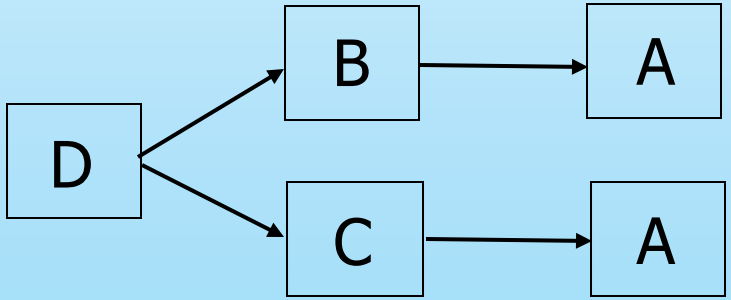
- Dependence of type ‘is-a’
- Reuse mechanism
  - Inherits and extends characteristics of already existing classes, refinement.
  - Existing class does not appear as data member of the new class
  - Base class (mother-class) and derived class (daughter class)
- Collect common characteristics in a base class
  - Mention a method in a base class and define it in a derived class
  - Pure virtual (abstract) methods, abstract classes

C++ Multiple Inheritance & Virtual Classes (1/2)

- Hierarchy
- From multiple base classes
  - Example

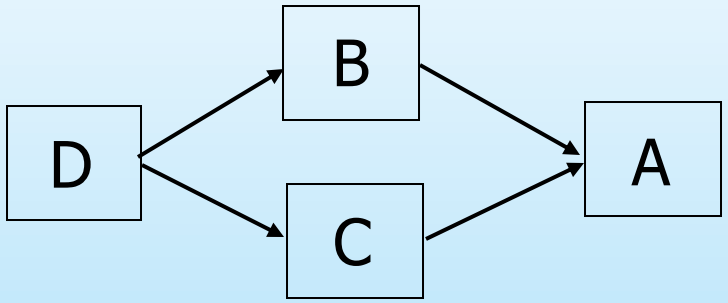


- Through multiple layers
  - Non-shared base classes, explicit naming
    - Example



C++ Multiple Inheritance & Virtual Classes (2/2)

- Through multiple layers
  - Virtual inheritance -> virtual (base) classes
    - Example



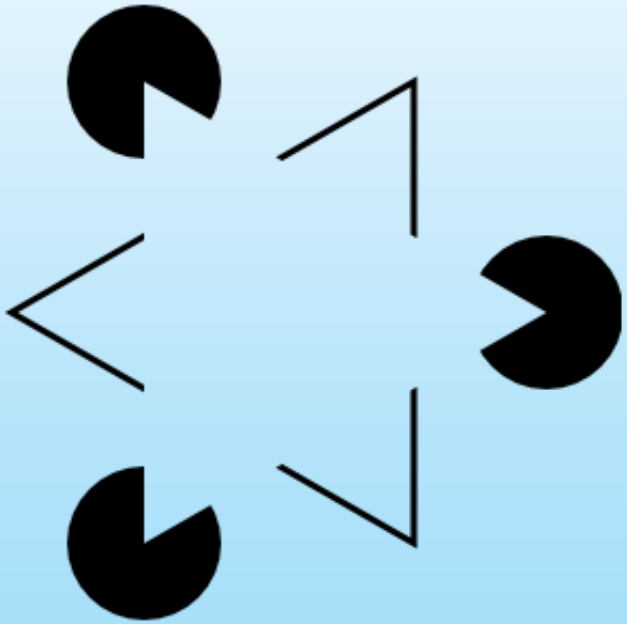
C++ Polymorphism (1/3)

- What's this ?

- Two ink patterns rotated … ummm, sure?

C++ Polymorphism (2/3)

- … if grouped conveniently
  - The ink patterns form a white triangle !



C++ Polymorphism (3/3)

- An object can take several forms
- Universal polymorphism
  - Object of a derived class is also object of the base class -> early/static binding -> right function selected at compiled-time -> method redefinition
  - Pointer to the base class also assumes the type of the object (of a derived class) it points to -> late/dynamic binding -> right function selected at run-time -> virtual methods
- Ad-hoc polymorphism
  - Different methods with the same name, different number and type of arguments (overload)
  - Data type conversions -> explicit/implicit

C++ Virtual Methods

- Late binding, Early binding and overload example

```
class Base {
 public:
     virtual void vMethodA();
     void vMethodB();
     virtual void vMethodC();
     void methodD();
};
class Derived:public Base {
 public:
     void vMethodA(); // late binding, implicitly virtual
     void vMethodB(int a);// overload
     float vMethodC(); // error: return type mismatch
     void methodD();   // early binding
};
```

Pure Virtual (Abstract) Methods / Abstract Classes

- Virtual methods in base class must be defined … Even if they are never used
- A pure virtual (abstract) method is a virtual method that has no implementation

```
class Base {
    virtual void pvMethodA() = 0; // pure virtual method
};
```

- A pure virtual method is defined in derived classes
- An abstract class has at least a pure virtual method
- No object instantiation of abstract class objects allowed
- Implicit inheritance of pure virtual methods

Late Binding & Interface Definition

- Late binding is responsible of interface definition
- Example

```
void main(){
  vehicle*  myVehicle;
  car*   myCar    = new car
  bike*     myBike   = new bike;
  // late binding
  myVehicle = myCar;  myVehicle->message();
  myVehicle = myBike; myVehicle->message();
}
```

C++ Composition & Utilization (1/2)

- Dependence of type ‘contains-a’, i.e. component of
- Data member(s) of the aggregated class(es) is(are) of the aggregating class type(s)
- Example:

```
class mouse{
   wheel  mainWheel;
   botton rightBotton, leftBotton;
};
```

- Dependence of type ‘use-a’, i.e client/server relationship
- Reference argument(s) of composed class method(s) is(are) of composing class type(s)

C++ Composition & Utilization (2/2)

- Example:

```
class driver{
  public
    void drive(car& myCar);
};
```

- Hands-on: C++ Exercises 1.a, 1.b, 1.c, 1.d

Outline of Previous Ideas

- Static declarations
- Operators
- Class templates
- Inheritance
- Polymorphism / Late binding
- Virtual Methods / Abstract classes
- Composition & Utilization