EEET2482/COSC2082

SOFTWARE ENGINEERING DESIGN, ADVANCED PROGRAMMING TECHNIQUES

WEEK 6 – INHERITANCE, VECTORS & RULES OF THUMBS

LECTURER: LINH TRAN

RMIT UNIVERSITY

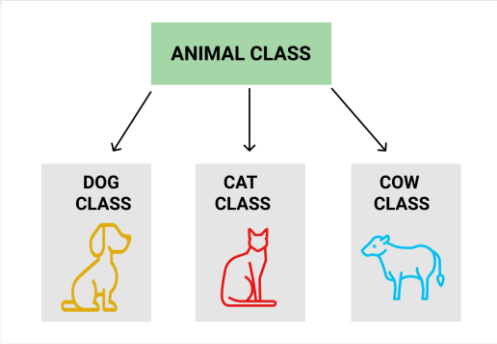
Review

- OOP principles
  - Classes and objects
  - Encapsulation
  - Constructor and Destructor
  - Friend Functions (sharing access)
- Passing/ access by value or reference.
- Operator Overloading: `+`, `-` , `++`, `--`, `<<`, `>>`
  - When to use a friend function ?
    - `Obj` + `Obj` Can use either a Class member or `friend` function
    - `Obj` + `int` Can use either a Class member or `friend` function
    - `int` + `Obj` must use a `friend` function (*first operand is not an object*)

```
class MyClass {
  private:
    int a;
  public:
    MyClass(int x) {a=x;}

    // External friend function
    friend void Double(MyClass *Ob);
};

void Double(MyClass *Ob) {
   cout << Ob->a * 2 << endl;
}

void main(){
   MyClass Ob(2);
   Double(&Ob);
}
```

Inheritance

A class can inherit attributes and methods from another class



Syntax:

```
class child_class: access_mode parent_class
```

- **base class** (parent) - the class being inherited from
- **derived class** (child) - the class that inherits from another class

Note: *besides inherited properties, the child class can has its own attributes and methods.*

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

// parent base class
class Animal {
public:
    void eat() {
        cout << "I can eat!" << endl;
    }
};

// child derived class
class Dog : public Animal {
public:
    void bark() {
        cout << "I can bark! Woof woof!!" << endl;
    }
};

int main() {

    Dog dog1;
    dog1.eat();  //dog1 has method from the parent class
    dog1.bark(); //dog1 has method from its own class

    return 0;
}
```

Output :

```
I can eat!
I can bark! Woof woof!!
```

Access Specifier *(when defining parent class)*

- A member (attribute/ method) of the base class can be accessible/inaccessible from a derived class due to its access specifier.
- Like `private` members, **`protected`** members are inaccessible outside of the class, however, they can be accessed by derived classes and friend classes/functions.

Access Specifiers	Within Same Class / From Friend Functions	From Child Derived Class	Outside Class
public	Yes	Yes	Yes
protected	Yes	**Yes**	No
private	Yes	**No**	**No**

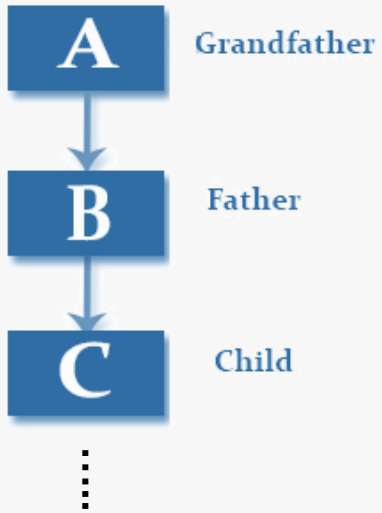
Access Mode *(when doing inheritance)*

- **`public` mode**: members of the base class are inherited by the derived class just as they are.
- **`protected` mode**: public members of the base class become protected in the derived class (others stay the same).
- **`private` mode**: all members of the base class become private in the derived class.

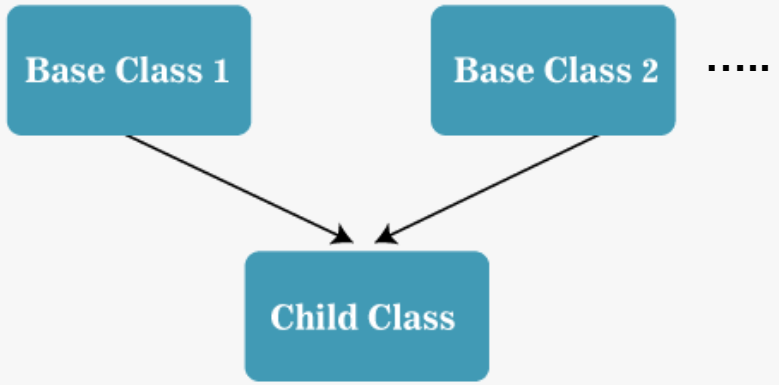
	Access mode of Inheritance		
Parent class member's access specifier	**public mode**	**protected mode**	**private mode**
public	public	**protected**	**private**
protected	**protected**	**protected**	**private**
private	**private**	**private**	**private**

Multilevel & Multiple Inheritance

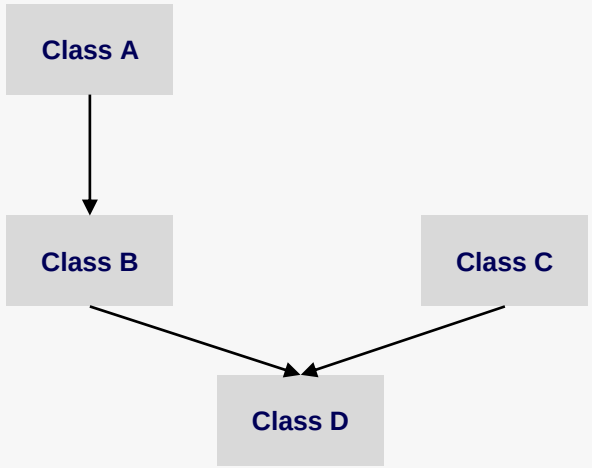
- **Multilevel Inheritance**: a class can also be *derived from one class, which is already derived from another class*.



- **Multiple Inheritance**: a class can also be derived from two or more base classes, using a **comma-separated list:**



Example




```
// Base class (grandfather)
class GrandFather {
public:
    void grandFunct() {
      std::cout << "A method from grandfather class \n" ;
    }
};

// Derived class (father)
class Father: public GrandFather {
};

// Another base class (mother)
class Mother{
public:
    void momFunct() {
      std::cout << "A method from Mother class \n" ;
    }

};
```

```
// Derived class (child)
class Child: public Father, public Mother {
};


int main() {
    Child myObj;
    myObj.grandFunct(); //The object has method from grandfather class
    myObj.momFunct();   //The object has method from Mother class
    return 0;
}
```

C++ Vectors

- Vectors are used to store **elements of a same data type**.
- However, unlike arrays, the size of a vector can grow dynamically *(its elements are allocated and stored in heap memory)*.
- Must include library header to use: `#include <vector>`
- Declaration Syntax:

```
std::vector<data_type> variable_name;
```

Useful Functions for Vectors

Function	
push_back()	Add element at the end
pop_back()	Delete last element
[n] at(n)	Access element at index n
front()	Accesss the first element
back()	Accesss the last element
begin()	Return iterator to beginning
end()	Return iterator to end
size()	Returns the number of elements present in the vector
insert()	Insert elements at a specific position
erase()	Erase some elements
clear()	Erase all the elements of the vector
empty()	Check if the vector is empty

- Initialization methods:

```
// Initializion with list (1-5)
std::vector<int> vectorA = {1, 2, 3, 4, 5};
std::vector<int> vectorB {1, 2, 3, 4, 5};

// Uniform initialization
// 5 elements with values are all 12
vector<int> vectorC(5, 12);
```

See: https://www.cplusplus.com/reference/vector/vector/
https://www.programiz.com/cpp-programming/vectors

Example

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> numVt {1, 2, 3, 4, 5};
    numVt.push_back(60); //add value 60 at the end
    numVt.push_back(70); //add value 70 at the end
    std::cout << "First element: " << numVt[0] << "\n";
    std::cout << "Last element: "  << numVt[numVt.size() - 1] << "\n";

    numVt.pop_back();    //remove last element
    std::cout << "Last element after removing: "  << numVt[numVt.size() - 1] << "\n";

    //Print out all elements with ranged based for loop
    std::cout << "All elements: ";
    for (int &num : numVt) {
        std::cout << num << " ";
    }

    return 0;
}
```

```
First element: 1
Last element: 70
Last element after removing: 60
All elements: 1 2 3 4 5 60
```

Constructor with member initializer list

```
class BankAcc{
public:
    std::string name;
    int amount;

    BankAcc() {} //Default Constructor

    /* Parameterized Constructor: Previous way*/
    BankAcc(std::string name_val, int amount_val) {
        name = name_val;
        amount = amount_val;
    }

    //Parameterized Constructor with member initializer list
    BankAcc(std::string name_val, int amount_val) :
        name(name_val), amount(amount_val) {}
};
```

Copy Constructor & Copy Assignment

- Assigning Objects:
  - Define and initialize a new object from an existing object: **Copy constructor** is called
  - Assign new value for an existing object from another existing object: **Copy Assignment operator** is called

- Copy Constructor:

```
ClassName (const ClassName &old_obj){
    //statements to copy content from old_obj to current_object
}
```

- Copy Assignment operator:

```
ClassName operator = (const ClassName &old_obj){
    //statements to copy content from old_obj to current_object;
    return *this; //return current object (so that multiple "=" operators can be used together)
}
```

C++ Rules of Thumbs

Approximately accurate guide or principles, based on experience or practice rather than theory.

- **Rule of three:** if a class requires a **user-defined destructor**, a user-defined **copy constructor**, or a user-defined **copy assignment operator**, it *almost certainly requires all three.*

- **Rule of five:** any class for which **move semantics** are desirable, has to declare all five special member functions (*move constructor, move assignment, user-defined destructor, copy-constructor, copy-assignment operator*).

- **Rule of zero**: any resource that needs to be managed should be defined in its own class type.

Move Semantics

40.1.4 Move Semantics

C++ 11 standard introduces the move semantics for classes. We can initialize our objects by moving the data from other objects. This is achieved through move constructors and move assignment operators. Both accept the so-called *rvalue reference* as an argument. *Lvalue* is an expression that can be used on the left-hand side of the assignment operation. *rvalues* are expressions that can be used on the right-hand side of an assignment. The rvalue reference has the signature of *some_type&&*. To cast an expression to an rvalue reference, we use the *std::move* function. A simple move constructor and move assignment signature are:

```
class MyClass
{
public:
    MyClass(MyClass&& otherobject) // move constructor
    {
        //implement the move logic here
    }

    MyClass& operator=(MyClass&& otherobject) // move assignment operator
    {
        // implement the copy logic here
        return *this;
    }
};
```