C++

Classes

CO650 Advanced Programming

Topics

- Classes
- Constructors
- Overloading
- Instantiation
- Access Modifiers
- Inheritance
- Abstract Classes
- Project Structure
- The UML Class

Terminology & Rules

- Functions declared within a class are known as Member Functions (methods in C# & Java).
- Variable declared within a class are known as Member Variables or Data Members.
- C++ allows the declaration of functions outside of a class

Defining Classes

The class keyword precedes the class's name which must conform to identifier naming conventions.

```
class  NameOfClass {


} optional variable/s;
```

Member Functions and Variables belonging to the class may be placed inside the { } Note the ; after the class's closing bracket.

```
class BasicPlayer {
        int size;
        void DoSomething() {
              cout << "Something" << endl;
        }
};
```

Functions defined inside the class are implicitly inline

Defining Member Functions

- Member functions may be defined outside of the class definition as long as they are preceded by the class name and scope resolution operator ::
- The class must include the function's declaration (prototype).

```
class NameOfClass{
    type MemberFunctionName();
};

type NameOfClass::MemberFunctionName(){
    // Statments
}
```

Defining Member Functions

```
class BasicPlayer {
        int size;
        void Hello() {
              cout << "Hello" << endl;
         }
         void GoodBye();
};


void BasicPlayer::GoodBye(){
              cout << "Good Bye" << endl;
}
```

The Constructor

- It is illegal to declare a member variable and initialise its value.

```
class Player {
        int number = 10;
};
```

- This generates a compiler error (in pre C++ 11 only).
- To overcome this we can initialise the class's member variables using a constructor.
- A constructor is a special member function that is automatically invoked when an object of the class is instantiated.

The Constructor

- The constructor has the same name as the class
- Unlike member functions it has no return type.

```
class  NameOfClass {
      NameOfClass(){
          // Statment/s
      }
};
```

- A Single class can contain multiple overloaded constructors

The Constructor

- Typically used to initialises member variables.

```
class Player {
public:
        int number;
        string name;
        Player(int pNumber, string pName){
              number = pNumber;
              name = pName;
        }
};
```

- Alternative syntax using an initialiser list

```
class Player {
public:
        int number;
        string name;
        Player(int pNumber, string pName):number(pNumber),name(pName){}
};
```

Overloading

- Often you may need more than one function with the same name.
- This is legal if they have either a different number of parameters or the parameter types are different.
- The return type may be the same

```
int Sum(int a , int b){
        return a + b;
}

float Sum(float a, float b){
                return a + b;
}

int main(){
        cout << Sum(3,5) << endl;
        cout << Sum(3.25f, 8.45f) << endl;
        return 0;
}
```

Instantiating an Object

- Constructors can’t be explicitly invoked
- They are executed when a new object of the class is instantiated.

```
ClassName identifier(parameter/s);
```

- The brackets after the variable name are optional if no arguments are passed.

```
int main(){
    BasicPlayer myBasicPlayer;
    Player myPlayer(7,"David Beckham");
    return 0;
}
```

Invoking member Functions

- The public member function can be invoked by placing a dot (object member access operator) after the object variable's name followed by the member function's name and the call operator ()

```
identifier.memberFunctionName();
```

```
Player myPlayer(7,"David Beckham");
myPlayer.DisplayName();
```

Access Modifiers

- Determine the visibility of members.
- **public** – visible to any member of any class
- **private** – accessible from within the same class
- **protected** – as private + its derived classes

```
class  NameOfClass {
private:
     one or more variable declarations or
     member functions
protected:
     one or more variable declarations or
     member functions
public:
     one or more variable declarations or
     member functions
};
```

Access Modifiers

- The order of the access modifiers is not important
- They can be repeated
- Constructors should have public access
- Default – i.e. no modifier reverts to private access

```
class Player {
private:
	int number;
	string name;
public:
	Player(int pNumber, string pName){
		number = pNumber;
		name = pName;
	}
	void DisplayName() {
		cout << "Player name is : " << name << endl;
	}
};
```

Structures

- Similar to C++ classes apart from two small differences
  - By default members in structure are public whilst the default in a class is private.
  - Inheritance in struct is public by default.
- Included with C++ to provide compatibility with C.

```
struct  NameOfStructure {


};
```

Creating Static Objects

- Classes can be regarded as templates from which we create (instantiate) objects.
- To create an object on the stack (static), first declare a variable of the class type

```
ClassName identifier;
ClassName identifier(parameter/s);
```

```
Player myPlayer;
```

- Invoke it’s member functions using the member selection operator . Followed by the function name and call operator ()

```
identifier.MemberFunctionName();
```

```
myPlayer.Print();
```

Creating Dynamic Objects

To create a dynamic object stored on the heap, first declare a pointer variable of the class type

```
ClassName *identifier;
```

```
Player *myPlayer;
```

Then create the object using the new keyword and assign it to the variable.

```
identifier = new ClassName();
```

```
myPlayer = new Player();
```

Invoke methods using the dereference operator

```
identifier->MemberFunctionName();
```

```
myPlayer->Print();
```

Example

```
class MyClass {
public:
        void Print() {
            cout << "MyClass  \n";
        }
};


int main(){
      MyClass myStaticObj;
      myStaticObj.Print();
      MyClass *myDynamicObj;
      myDynamicObj = new MyClass();
      myDynamicObj->Print();
      return 0;
}
```

Passing Objects As Pointers

Formal parameters can be pointers to objects

```
type functionName(ClassName * identifier)
```

When invoking the member function we can pass a pointer

```
ClassName *identifier = new ClassName();
functionName(identifier);
```

Example:

```
void Display(Player *p){
    cout << p->name << endl;
}
```

- Instantiate the player and pass it to Display

```
Player *myPlayer = new Player();
Display(myPlayer);
```

Creating Anonymous Objects

When passing pointers as arguments to a function it may not be necessary for the invoking code to maintain a pointer to the object.

functionName(new type)

```
class Weapon{
private:
	Magazine *magazine;
public:
	void Load(Magazine *newMag){
		magazine = newMag;
	}
};

int main(){
		Weapon *myGun = new Weapon("M14",true);
		myGun->Load(new Magazine(20));
		return 0;
}
```

Inheritance

- Typically classes have relationships with one another.
- Some classes being specialised versions of more general classes.
- This could be modelled as a class hierarchy.
- Examples

  dog is a type of animal

  house is a type of building

  npc is a type of gameObject

- The general class is known as the parent and the specialised class the child

```
class ChildClassName:ParentClassName{


}
```

Inheritance

```
class Model{
public:
        void Draw(){ }
};
```

```
class Car : public Model{
public:
        void ChangeGear(){ }
};
```

- The car above is a type of model
- When declaring the class we indicate car inherits the members of Model by including **: Model** after the class name.
- **public** is placed before the Model to ensure public member functions remain public when invoked on an object of the Car type. If Private was used all member functions inherited from Model would be private.
- Car is known as a derived (sub/child class) and Model the base class (super/parent class).

Inheritance

- Within the application we can create instances of both car and model.
- Model however can only invoke the Draw() member function.
- Car can invoke both its ChangeGear() and the model's Draw() member functions.

```
class Model {
public:
    void Draw(){}
};
```

```
class Car : public Model {
public:
    void ChangeGear(){}
};
```

```
int main(){
    Model model;
    Car car;
    model.Draw();
    car.Draw();
    car.ChangeGear();
}
```

Multiple Inheritance

- C++ supports multiple inheritance
- A derived class inherits from more than one parent class.
- Can introduce unnecessary complexity.
- Naming conflicts – Parents have members with the same name. Which should be used?
- Diamond problem – The two parent classes have a common parent.
- Multiple Inheritance should be avoided if possible.

```
class Boat {
protected:
        int id;
public:
        Boat(int id):id(id){}
        void Display(){ cout << id << endl; }
};

class Car {
protected:
        int id;
public:
        Car(int id):id(id){}
        void Display(){ cout << id << endl;}
};

class Amphibian :public Boat, public Car{     //Implement multiple inheritance
public:
        Amphibian(int id):Car(id),Boat(id) {}    //The amphibian has two id's!
};

int main() {
        Amphibian *duck = new Amphibian(1);
        duck->Car::Display();                    //Prefix Display with Class to avoid name conflict
        return 0;
}
```

Encapsulation

- Class member variables define an object's state.
- Direct access to the state could lead to inconsistencies and leads to tight coupling between the class and code that uses it.
- Good practice to hide this state by declaring members private. Forcing the developer to access the variables through getter / setter methods.
- There are two types of member functions.
- **Mutators** – allow changes to the object's state (default).
- **Accessors** – do not allow changes to the object's state.
- A member function can be declared an accessor by appending the const keyword to its declaration

```
type FunctionName() const { }
```

Getter & Setter member functions

- Each variable has a pair of member functions that sets and gets the variable value.
- Logic inside the setter, ensure the state remains consistent.
- The getter is an **Accessor** and consequently should not change the state – hence the addition of the **const** keyword.

```
class Marker {
private:
        int passMark;
public:
        void setPassMark(int pMark){
              if(pMark > 0)
                  passMark = pMark;
         }
         int getPassMark() const{
              return passMark;
         }
};
```

Abstract Member Functions

- Also know as **pure virtual member functions**.
- A function that has no implementation.
- It is virtual as subclasses must override it.

```
virtual type MemberFunctionName() = 0;
```

- When declared within a class it prevents the class from being instantiated (an abstract class).
- Only derived classes that implement the pure virtual member functions can be instantiated.

```
class Person {
public:
    virtual void Move() = 0;
};
```

```
class Soldier : public Person {
public:
    void Move(){ }
};
```

- Classes with one or more abstract member functions could be used as an alternative to Java / C# interfaces.

Interface Example

```
class IPrintable {
public:
    virtual void Print()=0;
};
```

While we can't create an instance of an abstract class, it is possible to define a pointer to the abstract class and assign a derived object to it.

This restricts access to the public members of the abstract class.

```
class GameObject: public IPrintable {
private:
    int id;
public:
    GameObject(int id):id(id){}
    int GetID() const {
        return id;
    }
    void Print(){
        cout << "ID: " << id << endl;
    }
};
```

```
int main(){
    GameObject *obj1 = new GameObject(1);
    cout << obj1->GetID() << endl;
    IPrintable *obj2 = new GameObject(2);
    obj2->Print();
}
```

Project Organisation

- Often a developer is only interested in the class's interface.
- Not the implementation of the member functions
- Separate the implementation and interface elements of a class.
- The **class's definition** (interface) including the declaration of its member functions is placed in a header file (ending in **.h**)
- The **definition** of the member functions is placed in a **.cpp** file.
- Within Visual Studio these files are created for you when you select **Project->Add Class**
- It is good practice to place each class in its own .h and .cpp file.
- By convention these files have the same name as the class. This is enforced when creating the classes through Visual Studio.

```
// File date.h
#pragma once

class Date {
private:
        int year;
        int month;
        int day;
public:
        Date(int day, int month, int year);
        int GetYear(){ return year;}
        int GetMonth(){ return month;}
        int GetDay(){ return day;}
        void SetDate(int d, int m, int y);
};
```

```
// File date.cpp
#include "StdAfx.h"
#include "Date.h"

Date::Date(int day, int month, int year){
        SetDate(day,month,year);
}


void Date::SetDate(int d, int m, int y){
        day = d;
        month = m;
        year = y;
}
```

```
#include "stdafx.h"
#include "Date.h"


int main(){
        Date *myDate = new Date(24,5,2010);
        return 0;
}
```

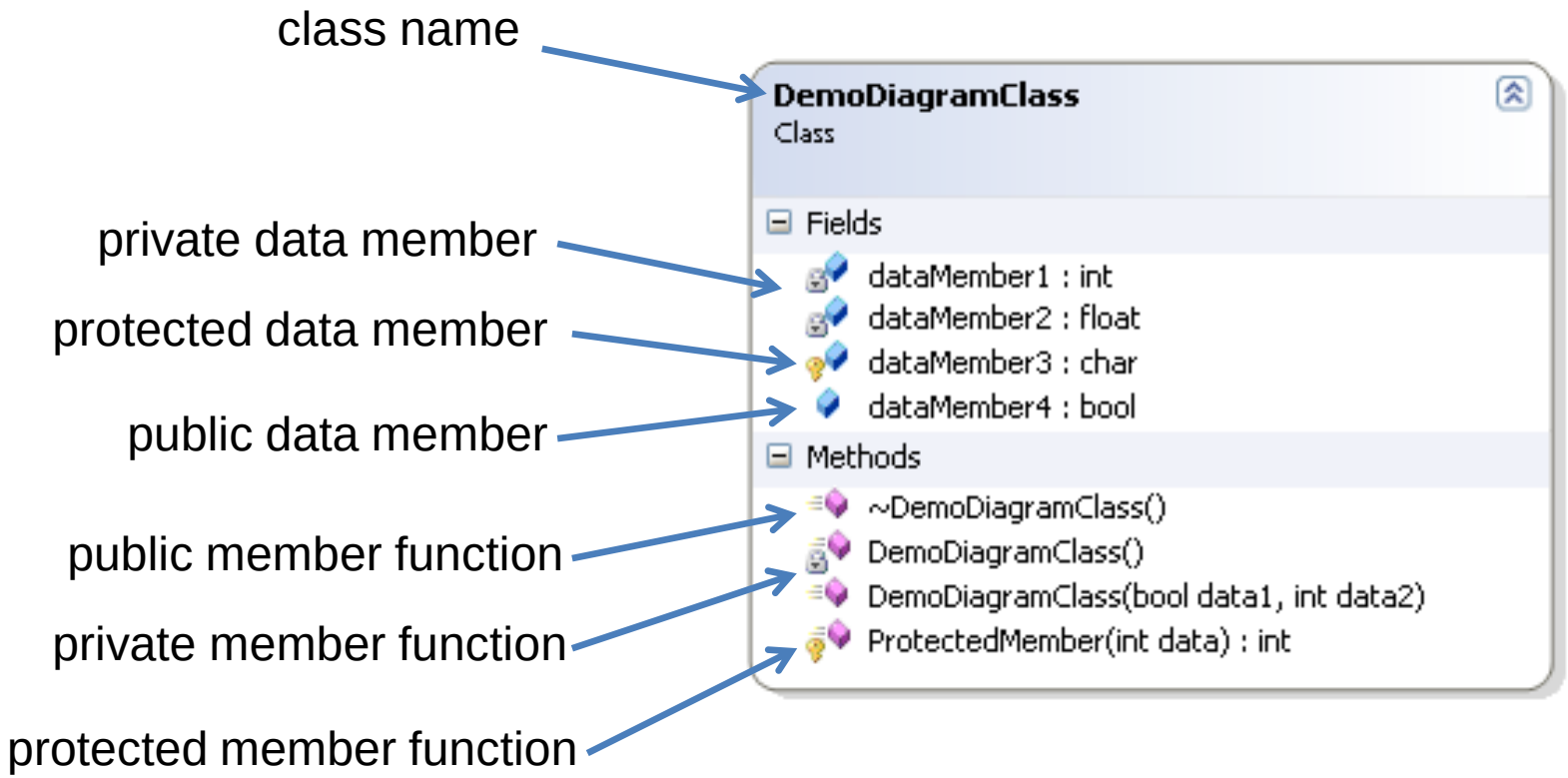
Project Organisation

- The header file should be included within any code that requires access to the class
- The #pragma once (O/S specific) directive ensures the header is only embeded within the project once. No matter how many times it is included.
- Often one line member functions are left inside the class's definition.

UML

- Many of the practicals contain class diagrams that I will ask you to implement.
- A basic understanding of how to interpret these diagrams is necessary.
- Rather than use an external package to create the diagrams, I have utilised Visual Studio's built in support for creating class diagrams.
- This ensures the diagram and code remain synchronised with changes in the code reflected in the diagram.

UML Class



General
padlock : private
key : protected
neither : public

Summary

- Classes are an important building block.
- Constructors are used for initialisation.
- Classes by conventions are placed in .h and .cpp files
- Use access modifiers to grant the least access possible.
- The const keyword can be applied in a number of different ways to stop side effects.