

C++ Polymorphism

CO650 Advanced Programming

Topics

- Arrays of Objects
- Polymorphism
- Dynamic Casting
- Friends

Static Arrays of Objects

- Creates an array of size elements of objType Type

```
type identifier[size];
```

- Objects are instantiated automatically.
- Does however require a default constructor (if a non default one has be defined).
- Invoke member functions using dot operator

```
identifier[index].MemberFunctionName();
```

Static Arrays of Objects Example

```
class GameObject{
public:
    int id;
    GameObject(){
    GameObject(int id):id(id){}
    void Display(){
        cout << "ID = " << id << endl;
    }
};

int main(int argc,char* argv[]){
    GameObject objArray[3];
    for(int n=0;n<3;n++)
        objArray[n].id = n;
    for(int n=0;n<3;n++)
        objArray[n].Display();

    return 0;
}
```

Dynamic Sized Arrays of Objects

- Where the size of the array is determined at runtime
- Declare a pointer of the array type

```
type *identifier;
```

- Create the array dynamically using the new operator

```
identifier = new type[size];
```

- Objects are instantiated automatically.
- Does however require a default constructor (if a non default one has be defined).
- Invoke member functions using dot operator

```
identifier[index].MemberFunctionName();
```

- Array is located on heap.

Dynamic Sized Arrays Example

```
class Group{  
private:  
    GameObject* objects;  
    int size;  
public:  
    Group(int n):size(n){  
        objects = new GameObject[size];  
        for(int n=0;n<size;n++)  
            objects[n].id = n;  
    }  
};
```

Static Arrays of Pointers to Objects

- Creates an array of size elements of pointers to objects of class type

```
type *identifier[size];
```

- Each element must be assigned a dynamically created object

```
identifier[index] = new ClassName();
```

- Invoke member functions

```
identifier[index]->MemberFunctionName();
```

- May not require a default constructor

Statically Sized Arrays Example

```
class Group{
private:
    GameObject* objects[3];
public:
    Group(){
        for(int n=0;n<3;n++)
            objects[n] = new GameObject(n);
    }
    void Display(){
        for(int n=0;n<3;n++)
            objects[n]->Display();
    }
};
```


Dynamic Arrays of Pointers to Objects

- The array name is a pointer to a pointer

```
type **identifier;
```

- Each element is a pointer

```
identifier = new type *[size];
```

- Each index must be dynamically assigned an object of the class type.

```
identifier[index] = new ClassName();
```

- May not require a default constructor
- Dereference *identifier

Dynamic Sized Arrays of Pointers Example

```
class Group{
private:
    GameObject** objects;
    int size;
public:
    Group(int n):size(n){
        objects = new GameObject*[size];
        for(int n=0;n<size;n++)
            objects[n] = new GameObject(n);
    }
    void Display(){
        for(int n=0;n<size;n++)
            objects[n]->Display();
    }
};
```

Arrays of Different Objects

An array can hold pointers to objects instantiated from different classes, as long as the array is declared with a common ancestor.

```
class Player:public GameObject{};

class NPC:public GameObject{};

class Group{
private:
    GameObject** objects;
    int size;
public:
    Group(int n):size(n){
        objects = new GameObject*[size];
        objects[0] = new GameObject();
        objects[1] = new Player();
        objects[2] = new NPC();
    }
};
```

Dynamic 2 Dimensional Arrays

- The array name is a pointer to a pointer

```
type **arrayName;
```

- Each element is a pointer

```
arrayName = new type *[size];
```

- Each element contains an array

```
for each element  
    arrayName[index] = new type[size];
```

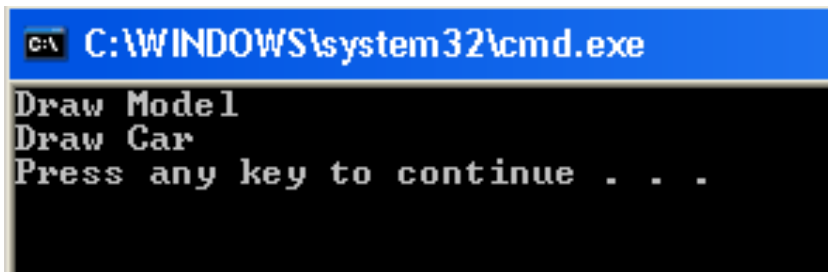
```
const int SIZE = 10;  
int **myArray;  
myArray = new int*[SIZE];  
for(int n=0;n<SIZE;n++)  
    myArray[n] = new int[SIZE];  
  
myArray[0][0] = 53;
```

Polymorphism

- The default C++ function invocations are statically bound.
- This means that during the compilation the invocation is matched to a corresponding function.
- Dynamic binding lets you match the invocation to a function at runtime.
- Dynamically bound functions are declared with the **virtual** keyword.
- Sub classes can define member functions with the same signature as those in their base class. When the member function is invoked on an instance of the sub class, it is the sub class function that is invoked not the base class method.
- If the Base class member function is labelled **virtual** then variables declared as type base class but assigned a sub class object will invoke the appropriate overridden sub class member function.

Polymorphism

Model's and Car's Draw method have the same signature.



```
C:\WINDOWS\system32\cmd.exe
Draw Model
Draw Car
Press any key to continue . . .
```

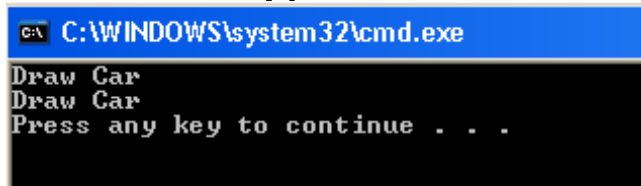
```
class Model{
public:
    void Draw(){
        cout << "Draw Model \n";
    }
};

class Car : public Model {
public:
    void Draw(){
        cout<< "Draw Car \n";
    }
};

int main(int argc,char* argv[]){
    Model *model = new Model();
    Car *car = new Car();
    model->Draw();
    car->Draw();
    return 0;
}
```

Virtual Functions

Model's Draw member function is **virtual**. This ensures objects of type Car will invoke the Car's Draw member function even though they might be stored within a pointer of the Base type.



```
C:\WINDOWS\system32\cmd.exe
Draw Car
Draw Car
Press any key to continue . . .
```

If Model's Draw member function was not virtual `model->Draw()` would display "Draw Model"

```
class Model{
public:
    virtual void Draw(){
        cout << "Draw Model \n";
    }
};
```

```
class Car : public Model {
public:
    void Draw(){
        cout<< "Draw Car \n";
    }
};
```

```
int main(int argc,char* argv[]){
    Model *model = new Car();
    Car *car = new Car();
    model->Draw();
    car->Draw();
    return 0;
}
```

Virtual Function Table

- A late binding mechanism consisting of a lookup table.
- Each class that has virtual functions or is derived from a class that has them has a vtable.
- The vtable is a static array populated at compile time each element containing a pointer to a virtual function that objects of the class can invoke.
- The compiler also adds a hidden pointer to the base class that points to the vtable for the base or sub class depending upon which has been instantiated.

Virtual Function Table

```
class Model{
public:
    VTable *_vptr;
    virtual void Update(){cout << "Update Model \n";}
    virtual void Draw(){cout << "Draw Model \n";}
};

class Car : public Model {
public:
    VTable *_vptr;
    void Draw(){cout<< "Draw Car \n";}
};

class Plane:public Model{
public:
    VTable *_vptr;
    void Update(){cout << "Update Plane \n";}
    void Draw(){cout<< "Draw Plane \n";}
};
```

Model vTable

Model::Update

Model::Draw

Car vTable

Model::Update

Car::Draw

Plane vTable

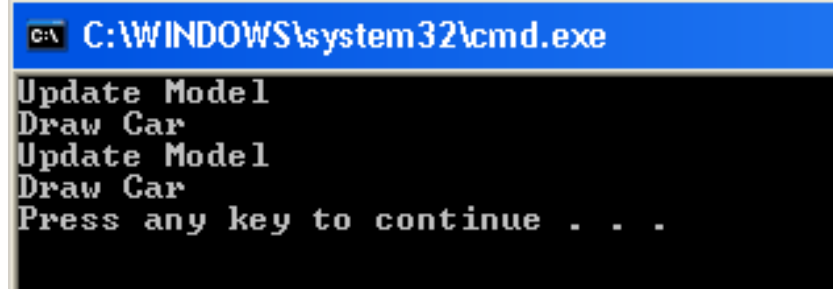
Plane::Update

Plane::Draw

- Note how the table only records the most derived functions accessible by the class.

Virtual Function Table

```
// carObj is a Car object and uses Car vtable
Car carObj;
carObj.Update();
carObj.Draw();
// pModel points to a Base object.
// However the _vptr to the vtable is in the base portion so *pModel points to the Car's vtable
Model *pModel = &carObj;
// It looks up the table entry for the Update and sees that it should invoke Model::Update()
pModel->Update();
// It looks up the table entry for the Draw and sees that it should invoke Car::Draw()
pModel->Draw();
```



A screenshot of a Windows command prompt window. The title bar is blue and contains the text "C:\WINDOWS\system32\cmd.exe". The command prompt itself has a black background with white text. The text displayed is:

```
Update Model
Draw Car
Update Model
Draw Car
Press any key to continue . . .
```

Virtual Function Table Performance

- Invoking a virtual function is slower than a non virtual function.
 1. The `*_vptr` is used to acquire the appropriate virtual table.
 2. The table must be traversed in order to find the correct function pointer.
 3. The function is invoked.

Override (C++11)

- Sometimes during maintenance the base and derived class member function become out of sync (adding new parameters).
- The optional **override** identifier explicitly states a function overrides a base virtual member function.
- If the base class virtual function doesn't exist the compiler will generate an error.

Override example

```
class GameObject{
public:
    virtual void Update(){
        cout << "Game Object Update" << endl;
    }
};

class Car:public GameObject{
public:
    void Update() override{
        cout << "Car Update" << endl;
    }
};
```

- In this example a compiler error will not be generated as the Car's Update has the same prototype as GameObject's.

Final (C++11)

- The final identifier stops derived classes overriding a member function.
- Then the final puts a 'floor' on how much overriding can be done in a class hierarchy.

```
class GameObject: Parent {  
public:  
    virtual void Register() final{};  
};
```

- When applied to a class it makes it non-inheritable.

```
class Immutable final{  
private:  
    int readOnlyData;  
    Immutable(int data):readOnlyData(data){}  
    int GetData(){return readOnlyData;}  
};
```

Base Class Access

- It is possible to access the members of a parent class through a child class.
- Prefix the member with the parent's class name

```
class Model{
public:
    virtual void Draw(){
        cout << "Draw Model \n";
    }
};

class Car : public Model {
public:
    void Draw(){
        Model::Draw();
        cout<< "Draw Car \n";
    }
};
```

Upcasting

- Converting a derived class pointer (ref) to a base class.
- Legal without explicit conversion as it conforms to is-a relationship.

```
class Computer{};

class Laptop:public Computer{};

int main(){
    Computer *computer = new Laptop;
}
```

- Virtual member functions (if defined) will be invoked on the child object.

Downcasting

- Converting base class pointer (ref) to derived class pointer (ref).
- Is-a rule may not be true. Child class may add new members.
- Must explicitly convert using a cast.

This example will **NOT** compile. No implicit conversion will take place

```
class Computer{};

class Laptop:public Computer{};

int main(){
    Laptop *laptop = new Computer;
}
```

dynamic_casting

- Checks it is safe to assign the address of an object to a pointer or reference.

```
dynamic_cast<type-id>(expression)
```

- Returns the address of the object if conversion is safe or 0 if not safe.

```
Car *car = dynamic_cast<Car*>(obj);
```

- Usually used when we have a pointer of a base type but want to perform derived class operations on the child object that is assigned to it.
- Classes must be polymorphic.

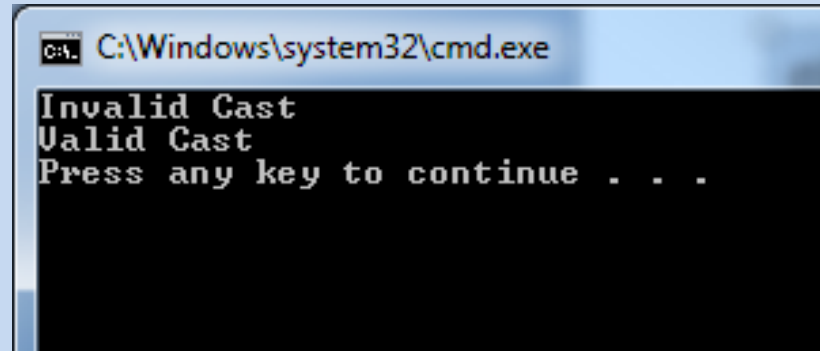
dynamic_casting Example

```
class GameObject{  
public:  
    GameObject(){}  
    virtual void Draw(){}  
};
```

```
class Car:public GameObject{  
public:  
    Car(){}  
};
```

```
void f(GameObject *obj){  
    Car *car = dynamic_cast<Car*>(obj); // Returns 0 if casting fails  
    if (car) cout << "Valid Cast" << endl;  
    else cout << "Invalid Cast" << endl;  
}
```

```
int main(){  
    f(new GameObject());  
    f(new Car());  
}
```



The screenshot shows a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The output of the program is displayed in white text on a black background:

```
Invalid Cast  
Valid Cast  
Press any key to continue . . .
```

Friend Functions

- Private and protected members can't be accessed from outside of the class / class hierarchy.
- An external function can be declared as a friend of a class, which allows it access to these class members.

```
class ClassName{  
    friend returnType functionName(parameter/s);  
}
```

- In the next example the friend function creates an array of all the unMounted weapons in a game. isMounted is a private member so the function is declared as a friend of the class.

Friend Function Example

```
class Weapon{
private:
    bool isMounted;
public:
    string name;
    Weapon(string desc,bool mounted):name(desc),isMounted(mounted){}
    friend Weapon** UnmountedWeapons(Weapon **,int* );
};

Weapon** UnmountedWeapons(Weapon **weaponList,int *size){
    Weapon **unMounted = new Weapon*[*size];
    int i = 0;
    for(int n=0;n < *size;n++){
        if (!weaponList[n]->isMounted){
            unMounted[i] = weaponList[n];
            i++;
        }
    }
    *size = i;
    return unMounted;
}
```

Friend Classes

- Classes can also be friends of other classes.
- Providing access to the private and protected members.

```
class ClassName{  
    friend class className;  
}
```

- Class B has access to class A's private & protected members.

```
class A{  
    friend class B;  
}
```

Friend Class Example

- In this example the UnmountedWeapons function has been made a member function of the WeaponsManager class (not show).
- The WeaponsManager class requires access to the private member isMounted and has therefore been made a friend of Weapon.

```
class Weapon{  
private:  
    bool isMounted;  
public:  
    string name;  
    Weapon(string desc,bool mounted):name(desc),isMounted(mounted){}  
    friend class WeaponsManager;  
};
```

Summary

- While C++ encourages the use of vectors as an alternative to arrays, an understanding of how arrays works is essential for legacy system maintenance.
- Polymorphism is a powerful tool used but does have a performance penalty.
- Think carefully about the need to use friend – is there a design flaw in your system?