C++

Constructors

CO650 Advanced Programming

Topics

- Constructors
- this
- References
- Copy Constructor
- Converting Constructors
- Static Members

Constructors

- Recall how we created a static object

```
MyClass myObject;
```

- myObject invokes a default constructor
- **Warning** : The following line declares a function !!

```
MyClass myObject();
```

- You can define your own constructor/s
- Typically they are used for initialising values.
- Constructor Rules
  1. Must have the same name as the class.
  2. Have no return type or statement.
  3. Usually made public

Default Constructors

- A default constructor is one that can be invoked without passing parameters.
- If you don’t define it, the compiler may create a default one for you within the executable.
- A Complier generated default constructors will allocate memory but not initialise values
- Below is an example of a user defined default constructor

```
 class MyClass{
private:
        int value;
public:
        MyClass(){
            value = 0;
        }
};
```

Default Constructors

- A default constructor declared with private access will prevent objects of that type being defined.

```
class MyClass{
private:
        int value;
        MyClass(){ }
};
```

This example will **NOT** compile. Can't access private member.

```
MyClass myObject;
```

Default Constructors

- A default constructor may have formal parameters as long as they are assigned default values.

```
class MyClass{
private:
        int value;
public:
        MyClass(int v = 0) {
                value = v;
        }
};
```

- This allows the constructor to be invoked without passing any arguments.
- A compiler error will be generated if you include both a default constructor with no parameters and a default constructor with parameters with default values.

Non Default Constructors

- A Single class may contain multiple overloaded constructors

```
class MyClass{
private:
        int a, b;
public:
        MyClass(int n){
                a = n;
                b = 0;
        }
        MyClass(int n, int m){
                a = n;
                b = m;
        }
};
```

- If a non default constructor is defined the compiler will not automatically create a default constructor.
- If a non default constructor is defined and no default constructor is defined the following will generate a compiler error.

```
MyClass myObject;
```

Invoking a Non Default Constructor

- Invoke a dynamic (heap) constructor

```
MyClass *myObject = new MyClass(20);
```

or static (stack)

```
MyClass myObject(30);
```

- Declaring an array of objects will require a default constructor as this is invoked automatically by the compiler

This example will **NOT** compile. As MyClass does not have a default constructor.

```
MyClass myObjects[5];
```

Initialising Data Members

- Unlike global variables data members will not be initialised with default values.
- If uninitialized their content is undetermined.
- Data members can’t be initialised when defined.

This example will **NOT** compile. Only static const data members can be initialised

```
class MyClass {
public:
        int id = 1;
};
```

- Normal practice would be to initialise the data members within the constructor.

```
class MyClass {
public:
        int id;
        MyClass(){
            id = 1;
        }
};
```

Member Initialiser

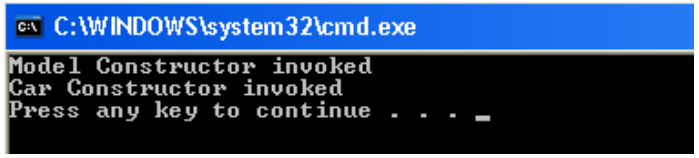
- Shorthand way of assigning values to data members.
- As if your primitive types had their own constructor.
- Constructor's parameter list is followed by a colon and a comma delimitated list of data members, with the value to be assigned in brackets.
- Should be included in the constructors **definition only**. Not the declaration.

ClassName(type param1, type param2) **:memVar1(param1), memVar2(param2);**

```
class MyClass {
private:
        int value;
public:
        MyClass(int theValue) : value(theValue) {
        }
};
```

Default Constructors & Inheritance

Constructors will implicitly call the default constructor of the base class.



First the base class constructor is invoked and then the derived class constructor.

```
class Model{
  public:
      Model(){
            cout << "Model Constructor invoked \n";
      }
};
```

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

```
int main(){
      Car *car = new Car();
      return 0;
}
```

Constructors & Inheritance

This example will **NOT** compile.

The compiler will try to invoke the Model's default constructor, which doesn't exist as the definition of the non default constructor stops the compiler generating a default constructor.

```
class Model{
private:
        int id;
public:
                Model(int modelID){
                    id = modelID;
                }
};
```

Note:

If a child class has a default constructor then the parent must also have one. Even thought the child constructor may not explicitly be used.

```
class Car:public Model{
public:
                Car(int carID){
                }
};
```

```
int main(){
      Car *carB = new Car(3);
}
```

Constructors & Inheritance

I've added a default constructor to the parent class, so the project will now compile.

In this example the argument 3 passed to the Car's constructor will be lost (not assigned to the id data member defined within Model).

```
class Model{
private:
        int id;
public:
                Model(){}
                Model(int modelID){
                        id = modelID;
                }
};
```

```
class Car:public Model{
public:
                Car(int carID){ }
};
```

```
int main(){
      Car *carB = new Car(3);
}
```

Constructors & Inheritance

- Constructors can be invoked from within the derived classes constructor as part of the initialisation list.
- The parent's constructor name follows the colon after the child's Constructor.
- Arguments can be passed to the parents constructor.
- In this example the carID parameter that is assigned a value of 3 is passed up to the Model's constructor.
- The invocation of Model's non default constructor means that Model's default constructor is redundant.

```
class Model{
private:
        int id;
public:
                // Model(){}
                Model(int modelID){
                        id = modelID;
                }
};
```

```
class Car:public Model{
public:
                Car(int carID):Model(carID){
                }
};
```

```
int main(){
      Car *carB = new Car(3);
}
```

Default Specifier (C++11)

- By default, C++ will provide a default constructor, copy constructor, copy assignment operator (=) and a destructor.
- These will not be generated if non default versions exist.
- default instruct the compiler to generate a default constructor. Regardless of the exist of non default versions.
- Not implemented in VS 2012.

```
class GameObject{
private:
	int id;
public:
	GameObject(int id):id(id){}
	GameObject() = default;
};
```

this

- this is a pointer to the current object
- Can only be used within the Class member functions
- this is passed as an implicit parameter to the member functions

```
class MyClass {
private:
        int value;
public:

        MyClass(int value) {
                this->value =  value;
        }
};
```

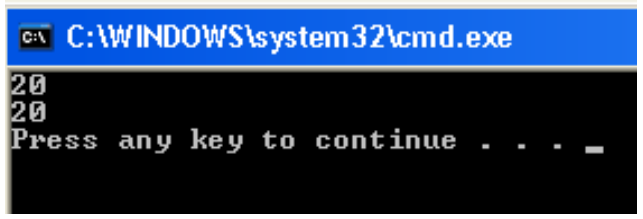
- this can be used to differentiate data members and parameters with the same name.

C++ References

- An alias for a variable
- The variables value can be changed through the reference
- Commonly used to pass objects as arguments to functions, where the function changes the state of the argument.
- A reference type has an **&** after its type specifier
- Apart from parameters and return values all references must be initialised when they are defined (initialising to null/0 is not allowed).
- Once defined a reference can't be changed. Although the value it references can.

Parameters (By Reference)

- The static cmd object is created within main and passed to the NPC's Perform member function, which accepts a reference (Command&).
- The reference is an alias for the argument passed, so the cmd variable in the main is referring to the same command object as the c parameter in Perform.




```
class Command{
public:
    int id;
    Command(int id):id(id){}
    void Print() { cout << id << endl;}
};

class NPC {
public:
     void Perform( Command& c){
                c.Print();
     }
};
```

```
int main(){
   Command cmd(20);
   NPC npc;
   cmd.Print();
   npc.Perform(cmd);
   return 0;
}
```

Copy Constructors

- Initialises an object with the values of another object of the same type.
- Invoked automatically by the compiler during initialisation operations.
- Like the default constructor the compiler will generate one automatically.
- Default behaviour results in the non static members of one object, copied to the other.
- Pointer data members have their address copied, so still share the same value!
- User defined copy constructors can override the default behaviour.

Copy Constructor

- Copy constructor is a member function with the same name as the class. It takes a single argument (usually const) that is a reference to an object of the same class.

```
class MyClass{
public:
    MyClass(const MyClass& obj){ }
};
```

- The copy constructor is invoked when
  1. Initialise one object from another of the same type
  2. Copy an object to pass as an argument to a formal parameter
  3. Copy an object to return it from a function
  4. Initialise the elements in a sequential container
  5. Initialise the elements in an array when an initializer list is used.

Invoking Copy Constructor

```
class MyClass{
public:
      MyClass() { }
      MyClass(MyClass& obj){
            cout << "Copy constructor invoked" << endl;
       }
       void DoSomething(MyClass obj){}
       MyClass ReturnAnObject(){
            MyClass obj;
            return obj;
       }
};

int main(){
     MyClass obj1,obj2;
     MyClass obj3(obj1);          // Invokes copy constructor
     obj3 = obj1;                 // Doesn't invoke copy constructor - only initialisation
     MyClass obj4 = obj2;         // Invokes copy constructor
     obj2.DoSomething(obj1);  // Invokes copy constructor
     obj2.ReturnAnObject();     // Invokes copy constructor
}
```

Shallow vs Deep Copies

- The behaviour of the default copy constructor creates a **shallow copy**.
- All data members are copied, including pointers.
- The dynamical allocated memory pointed to by the pointer will not be copied.
- The pointers in both the original and the copied objects will point to the same memory location.
- A **deep copy** will dynamically allocate memory to ensure that pointers within the two objects do not reference the same memory location.
- Classes requiring a deep copy would typically implement a destructor to free the dynamic memory and overload the assignment operator. Both topics to be covered later.

Copy Constructors

- The behaviour of the default copy constructor can be overridden by defining a copy constructor.
- If defined, it will be invoked when one class object is initialised with another.
- The copy constructor is declared with one formal parameter, a reference to an object of the class type.

```
ClassName (const ClassName &identifier):
```

- The object to be copied is passed as a constant to ensure that the members can’t be changed.
- The next example replicates the behaviour of the default copy constructor by creating a shallow copy.
- While the second creates a deep copy.

Shallow Copy Example

```
class ShallowClass {
public:
        Data *value;
        ShallowClass(){
                value = new Data(0);
        }
        ShallowClass(const ShallowClass& c) {
                this->value = c.value;
        }
};

int main(){
        ShallowClass obj1;   // Will fail if no default constructor exists
        ShallowClass obj2(obj1);
        ShallowClass obj3 = obj2;
        obj1.value->data = 25;
        cout << "Obj1.value->data : " << obj1.value->data << endl;    // Outputs 25
        cout << "Obj2.value->data : " << obj2.value->data << endl;    // Outputs 25
        cout << "Obj3.value->data : " << obj3.value->data << endl;    // Outputs 25
        return 0;
}
```

```
class Data{
public:
    int data;
    Data(int d):data(d){}
};
```

Deep Copy Example

```
class DeepClass {
public:
        Data *value;
        DeepClass(){
                value = new Data(0);
        }
        DeepClass(const DeepClass& c) {
                this->value = new Data(c.value->data);
        }
};

int main(){
        DeepClass obj1;
        DeepClass obj2(obj1);      // Copy constructor invoked
        DeepClass obj3 = obj2;     // Copy constructor invoked
        obj1.value->data = 25;
        cout << "Obj1.value->data : " << obj1.value->data << endl;   // Outputs 25
        cout << "Obj2.value->data : " << obj2.value->data << endl;   // Outputs 0
        cout << "Obj3.value->data : " << obj3.value->data << endl;   // Outputs 0
        return 0;
}
```

```
class Data{
public:
    int data;
    Data(int d):data(d){}
};
```

Delete Specifier

- Prevents a function from being defined or invoked.

This example will **NOT** compile. As the copy constructor can't be invoked.

```
class GameObject{
public:
      GameObject(const GameObject&) = delete;
};

int main(){
    GameObject obj1;
    GameObject obj2(obj1);
}
```

Converting Constructor

- A constructor with a single parameter or multiple parameters with defaults values apart from the first, is known as a converting constructor.
- Can be used to implicitly convert when declaring and initialising and object.

```
class Conversion{
private:
        int a;
public:
        Conversion(int value):a(value){
                cout << a << endl;
        }
};


int main(){
        Conversion conv = 25;
}
```

Converting Constructor

- This can lead to confusion so to turn this behaviour off place the explicit keyword in front of the constructor.

This example will **NOT** compile.

```
class Conversion{
private:
        int a;
public:
        explicit Conversion(int value):a(value){
                cout << a << endl;
        }
};


int main(){
        Conversion conv = 25;
}
```

Static Data Members

- Static members can be either variables or functions.
- Static variables (class variables) belong to the class. A single value for all objects instantiated from the class.
- Similar to global variables but within the scope of the class and must obey access modifier rules.

```
static type identifier;
```

```
class Player{
public:
    static int instances;
    Player(){
        instances++;
    }
};
```

- Static data members are not initialised in the constructor.

Static Data Members

- While the static variable is declared within the class definition it **must** be defined (initialised) outside the class. This avoids multiple initialisations.

```
type ClassName::identifier = value;
```

- Avoid placing the definition inside a header file as this may lead to a multiple definitions error. Place it in the cpp or program file.

```
class Player{
public:
        static int instances;
        Player(){
            instances++;
        }
};
```

```
 int Player::instances = 0;

int main(){
}
```

Static Data Members

- The class variable can be accessed either through the object or the class.

```
ClassName::identifier
ObjectName.identifier
ObjectName->identifier
```

```
int main(){
    Player *player1 = new Player();
    cout << Player::instances << endl;
    cout << player1->instances << endl;
    return 0;
}
```

Static Members Functions

- Can only access static data members.
- Does not have a **this** parameter.
- The declaration includes the static keyword.

```
static type Identifier(parameter/s);
```

- If defined outside of the class the static keyword should be dropped from the definition.

Static Members Functions

```
class Player{
public:
        static int instances;
        Player(){
                instances++;
        }
        static void PrintStaticValues(){
                cout << instances << endl;
        }
};

int Player::instances = 0;

int main(){
        Player *player1 = new Player();
        player1->PrintStaticValues();
        Player::PrintStaticValues();
        return 0;
}
```

Summary

- Constructors are used to initialise data members.
- The logic associated with copying objects can be implemented through the copy constructor.
- Static members provide a means of implementing encapsulated global variables and constants.