Object Oriented Methodology

Week – 3, Lecture – 2

Classes in C++ - Part 1

(Constructors, Destructors and Access Control)

SAURABH SRIVASTAVA

VISITING FACULTY

IIIT LUCKNOW

Creating classes in C++

You can either create a class in the *default* namespace, or use a specific namespace

- Remember, a namespace is like a collection of classes, structures, functions etc.

Do not be confused by the `using` keyword while defining your class

- While reading examples, you may read the following statement often in books or tutorials:
  `using namespace std;`
- This *does not* mean that the code in the file belongs to the namespace `std`
- It means, "make everything from the namespace `std` visible below"
- For example, it allows you to write "`cout`" instead of "`std::cout`"

Creating a class is similar to creating a structure template

The functions of a class, also called methods, are usually only declared in the class template

- They are defined outside the class, often in a different file

Example – A `Toy` Class (1/4)

```
#ifndef __Toy
#define __TOy

#include<string>

using std::string;

namespace example
{
        class Toy
        {
                private:
                        string name;
                        int price;
                        int id;

                        static int counter;

                public:
                        Toy(string, int);
                        ~Toy();
                        void start_playing();
                        void stop_playing();
                        string get_name();
                        void set_price(int);
                        int get_price();
                        int get_id();
        };
}
#endif
```

Example – A `Toy` Class (1/4)

```
#ifndef __Toy
#define __TOy

#include<string>

using std::string;

namespace example
{
        class Toy
        {
                private:
                        string name;
                        int price;
                        int id;

                        static int counter;

                public:
                        Toy(string, int);
                        ~Toy();
                        void start_playing();
                        void stop_playing();
                        string get_name();
                        void set_price(int);
                        int get_price();
                        int get_id();
        };
}
#endif
```

Here, we are creating a new class called `Toy` in the namespace `example`

The class has four fields, with one of them being *static* (more on this in the next lecture)

The class has eight methods, with two of them being special methods, called a constructor and a destructor (more on this soon)

It is a good practice to provide a class declaration in a header file, with the definitions for the method provided in a CPP file

Example – A `Toy` Class (2/4)

```
void Toy::start_playing()
{
        cout<<"You are now playing with "<<name<<" with id "<<id<<endl;
}

void Toy::stop_playing()
{
        cout<<"The "<<name<<" with id "<<id<<" is lying idle"<<endl;
}

string Toy::get_name()
{
        return name;
}

int Toy::get_price()
{
        return price;
}

void Toy::set_price(int i)
{
        price = i;
}

int Toy::get_id()
{
        return id;
}
```

Example – A `Toy` Class (2/4)

```
void Toy::start_playing()
{
        cout<<"You are now playing with "<<name<<" with id "<<id<<endl;
}

void Toy::stop_playing()
{
        cout<<"The "<<name<<" with id "<<id<<" is lying idle"<<endl;
}

string Toy::get_name()
{
        return name;
}

int Toy::get_price()
{
        return price;
}

void Toy::set_price(int i)
{
        price = i;
}

int Toy::get_id()
{
        return id;
}
```

The methods can be defined (usually in a .CPP file) like this…

The syntax is:
<return type> <class>::<method>(<arguments>)

The code is written in a code block with the namespace `example` (similar to the class declaration)

Access control over members of a class

You may have noticed that the fields were declared in the `private` section of the class

All the fields defined in the `private` section are kind of “invisible” outside the class

- So, they cannot be accessed outside the class by simply using `<object>.<member>` mechanism
- Although, it is common to keep the fields of a class private, methods too can be made private

The `public` members of a class are ”visible” to the outside world

- Thus, they can be accessed by using the `<object>.<member>` mechanism
- Although, usually the methods of a class are kept public, you are free to make fields public too

If you do not create these access control sections, by default, all members become private

The definitions of private methods outside the class’s declaration though is valid

Example – A `Toy` Class (2/4)

```
void Toy::start_playing()
{
        cout<<"You are now playing with "<<name<<" with id "<<id<<endl;
}

void Toy::stop_playing()
{
        cout<<"The "<<name<<" with id "<<id<<" is lying idle"<<endl;
}

string Toy::get_name()
{
        return name;
}

int Toy::get_price()
{
        return price;
}

void Toy::set_price(int i)
{
        price = i;
}

int Toy::get_id()
{
        return id;
}
```

The methods can be defined (usually in a .CPP file) like this…

The syntax is:
<return type> <class>::<method>(<arguments>)

The code is written in a code block with the namespace `example` (similar to the class declaration)

The associated access type (`private` or `public`) doesn't matter when you define the body of a method like this

Constructors and Destructors (1/2)

Constructors are methods which are invoked when a new object of a class is to be initialised

- It has the same name as the name of the class, without any return type (not even `void`)

A *default* constructor is added by the compiler for you, if you do not define one explicitly

- This added constructor does not have any instructions though – you can assume it to have empty body
- All the fields of the object are thus, initialised to their respective defaults

A class can have one or more constructors ...

- However, in case of multiple constructors, each constructor must have a different argument list

When you create a new object, you can pick any one of the constructors to initialise the fields

- For example, the statements, `A ob = A();` and `A ob = A(5)`, invoke different constructors
- The former invokes one with no arguments, and the latter invokes one with an `int` argument

If you define even a single constructor, the compiler doesn't add the default constructor

Example – A `Toy` Class (3/4)

```
int Toy::counter = 1;

Toy::Toy(string s, int i)
{
        id = counter++;
        name = s;
        price = i;
        cout<<"Creating a new "<<name<<" with id "<<id<<endl;
}

Toy::~Toy()
{
        cout<<"Destroying the "<<name<<" with id "<<id<<endl;
}
```

This is an example of a constructor

The constructor takes two inputs, and then assign it to two fields of the newly created object

It assigns a value to another field (`id`) via some other mechanism

Remember that since we have provided a constructor for the class, the compiler doesn't add any default constructor

Constructors and Destructors (2/2)

A destructor is a method that is called right before an object is “thrown into trash”

- It is not literally thrown, it means the memory allocated for the object is reclaimed

Its name is the name of the class, preceded by a tilde (`~`), without any return type (not even `void`)

A destructor can be used to perform any “clean-up” before the object becomes inaccessible

- For instance, if we allocated any memory dynamically, say via `malloc()`, it should be freed

A *default* destructor with empty body is added to a class automatically if it is not defined explicitly

While you can provide multiple constructors in a class, there can only

Example – A `Toy` Class (3/4)

```
int Toy::counter = 1;

Toy::Toy(string s, int i)
{
        id = counter++;
        name = s;
        price = i;
        cout<<"Creating a new "<<name<<" with id "<<id<<endl;
}

Toy::~Toy()
{
        cout<<"Destroying the "<<name<<" with id "<<id<<endl;
}
```

This is an example of a destructor

We are just printing a message here, so that we can know when an object is being "reclaimed"

Example – A `Toy` Class (4/4)

```
using std::string;
using std::cout;
using example::Toy;

void func()
{
        Toy t3 = Toy("Guitar", 349);
        t3.start_playing();
        t3.stop_playing();
}

int main()
{
        Toy t1 = Toy("Blocks", 399);
        Toy t2 = Toy("Rings", 499);
        func();
        return 0;
}
```

```
Creating a new Blocks with id 1
Creating a new Rings with id 2
Creating a new Guitar with id 3
You are now playing with Guitar with id 3
The Guitar with id 3 is lying idle
Destroying the Guitar with id 3
Destroying the Rings with id 2
Destroying the Blocks with id 1
```

Example – A `Toy` Class (4/4)

```
using std::string;
using std::cout;
using example::Toy;

void func()
{
        Toy t3 = Toy("Guitar", 349);
        t3.start_playing();
        t3.stop_playing();
}

int main()
{
        Toy t1 = Toy("Blocks", 399);
        Toy t2 = Toy("Rings", 499);
        func();
        return 0;
}
```

```
Creating a new Blocks with id 1
Creating a new Rings with id 2
Creating a new Guitar with id 3
You are now playing with Guitar with id 3
The Guitar with id 3 is lying idle
Destroying the Guitar with id 3
Destroying the Rings with id 2
Destroying the Blocks with id 1
```

On executing the code on the left, you will see the above output

The objects `t1` and `t2` are created before `t3`, but are destroyed after it – because the scope of `t3` is only within `func()`

Among the local variables, the objects are destroyed in the reverse order of their construction

Homework !!

For the class `Toy`, attempt to create an object without supplying the values for the parameters

- What can you infer from your observations?

Comment the supplied constructor in the class, and retry the above

Can you try to guess why the local objects are destroyed in the reverse order of their creation?

- **Hint**: It may have something to do with the use of a Stack frame for execution of a method