UNIVERSITY OF LIVERPOOL

Application Development with C++ (ELEC362)

Lecture 7: More classes

mihasan@liverpool.ac.uk

Previous lecture

- The definition of Object Oriented Programming and its 4 major components were discussed.
- Defining classes and objects in a code were discussed.
- The role of access specifiers was discussed.
- Structures were discussed and compared to classes.

This lecture

- What is covered in this lecture?

  1. Constructors and Destructors of classes 2. Classes and dynamic memory.

- Why it is covered?

  1. Constructors and Destructors are one of the core concepts in classes.

  2. Using dynamic memory in classes allow efficient code implementation in terms of memory.

- How are topics covered in this lecture:

  6 source codes

Classes revisited

- Encapsulation requires member data in a class to be private.
- Consider the following class:

```
class Box{
private:
     double length, width, height; // Member data
public:
     double boxVolume () {return height*width*length;}
     double boxSurfaceArea () {return 2*(height*width+height*length+length*width);}
     }
```

- Using an object from this class will either fail to compile, cause the code to crash or cause unknown behaviour.

Class constructor

- Definitions under `private` can only be accessed from within the class itself (i.e. only member functions or variable see them).
- They have to be initialised!! But we can't do that from outside the class because they are private.
- This problem is solved by defining a member function known as the constructor.
- The constructor is member function that is automatically called when an object is created to initialise data members of the object.
- It has the same name as the class and has no return type!
- Go to L7D1.cpp

Initialisation list

- Private data members can be set by constructors using initialisation lists:

```
class Ratio{
private:
        int num, den;
public:
        //Ratio(int n,int d) {num = n; den = d;};
        //or
        Ratio(int n,int d):num(n),den(d){//anything else here};
}
```

- Just like any other function, class constructors can be overloaded.
- Go to L7D2.cpp

Copy constructors

- Occasionally one might need copying an object, i.e.:

```
Ratio a(3,4); // normal constructor is called here
Ratio b = a; // b is a copy of a (copy constructor is called here)
```

- When an object is copied, the copy constructor is called.
- Copy constructor is similar to the constructor, it only differs in its parameter.

```
class Ratio{
private:
     int num, den;
public:
    Ratio(int n,int d) {num = n; den = d;};
    Ratio(const Ratio& r) {num = r.num; den = r.den;}; //Copy constructor
}
```

Must be a reference to prevent infinite chain of calls

Default constructor and explicit constructor

- If no constructor in a class is defined, the compiler generates a default constructor which does nothing!!

```
Ratio (){}; // Default constructor (called when no constructor is defined)
```

- The compiler uses implicit casting when calling single-argument constructors:

```
class Ratio{
private:
     int num, den;
public:
    Ratio(int n,int d) {num = n; den = d;};
    Ratio(int n) {num = n; den = 1;};                    }
Ratio a(2); // This is OK
Ratio b = 2; // This is also OK (despite the use of "=")
```

Explicit constructor

- To prevent the compiler from making implicit casting, the keyword "**explicit**" is used in the constructor's definition.

```
class Ratio{
private:
     int num, den;
public:
    Ratio(int n,int d) {num = n; den = d;};
    explicit Ratio(int n) {num = n; den = 1;};                    }


void main () {
Ratio a(2); // This is OK
Ratio b = 2; // Error: Not OK but b = Ratio(2) is OK
}
```

Class destructor

- The destructor is member function that is automatically called when an object is deleted or is out of scope.
- The destructor is primarily used to free dynamic memory after an object is deleted.
- It has the same name as the class's name preceded by ~.
- The default destructor does nothing.
- Go to L7D3.cpp

Static class members

- A static data member is a common member among all objects of the same class.
- Alternatively, a static data member belongs to the class rather than the object.
- A data member is made static by preceding its definition with keyword “static”.
- Static data members must be initialised when they are declared.

```
class Ratio{
private:
      int num, den;
public:
       static int counter = 0;
       Ratio(int n,int d): num(n),den(d){counter++};
} // The number of created objects is automatically counted
```

Friend functions

- In some codes, a function might be needed to work with multiple classes.
- To avoid repeating the definition of such function, or to allow the function to work with multiple classes at the same time. It is possible to create a stand-alone function with access to private members, by being defined as a "**`friend`**" function.
- Only the function's declaration/prototype is defined in the class preceded by the keyword "**`friend`**".
- Go to L7D4.cpp

Arrays of objects

- Just like any other datatype, it is possible to define arrays of objects.
- Example:

```
Ratio r1(5,6), r2(3,4);
Ratio r3(8,9);
Ratio Sequence[]{r1,r2,r3};
std::cout << Sequence[0].Fraction() << endl;
std::cout << Sequence[1].num << endl; // Error but why?
```

- Dynamic arrays of objects can be defined as well.

Objects and dynamic memory

- Just like other datatypes, objects can be defined on dynamic memory.
- It is possible to define pointers to objects, and to call objects by reference.

```
Ratio r1(5,6), r2(3,4);
Ratio *pr1{nullptr};
pr1=&r1; // Pointer to r1
Ratio &rr2(r2); // rr2 is a reference to r2
```

- Using a pointer to an object, it is possible to access member data and member functions using the arrow operator "->".

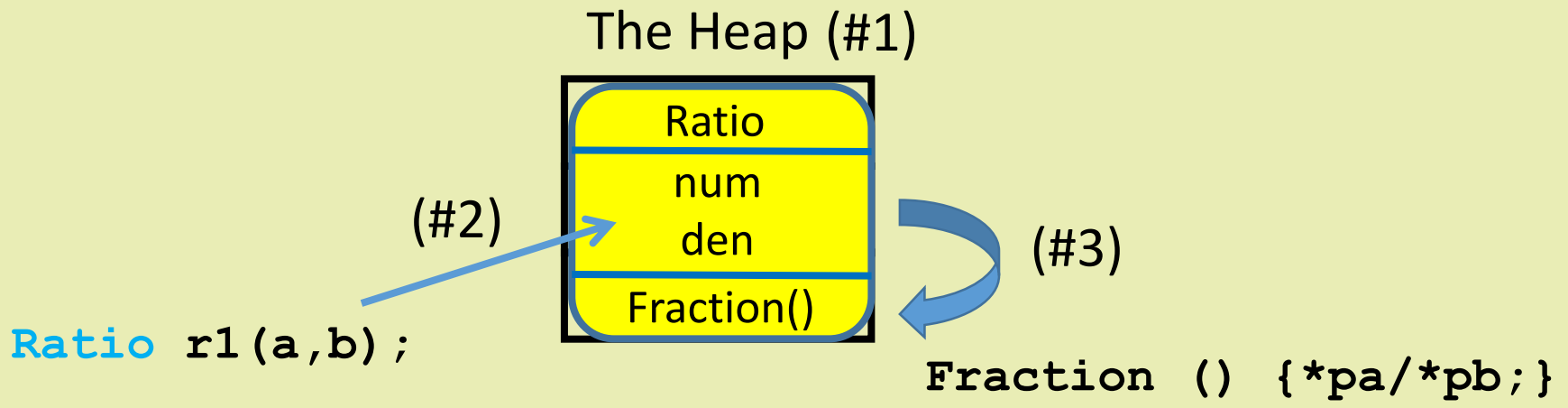
```
Ratio r1(5,6), *pr1{&r1};
pr1->Fraction(); // Calls the member function using a pointer
r1.Fraction(); // Calls the member function using the obj's name
```

Objects and memory management

- Objects can be memory-efficient in three ways:

1. By defining objects on the heap using "**new**" and "**delete**".
2. By using pointers/reference to pass parameters to initialize objects.
3. By using pointers/reference to pass parameters between member data and member functions.

- Go to L7D5.cpp



Practical note: Maximise the use of dynamic memory in your code.

“`this`” pointer

- Every object of a given class has its own data members while all objects share the member functions of the class.
- When a member function is called, how does it know which object it is dealing with?
- The pointer “`this`” hold the memory address of the object making the function call.
- The pointer is used implicitly whenever a function is called.
- The pointer “`this`” is used to access functions from the inside of the class, while standard pointers are used to access function from outside the class.
- Go to L7D6.cpp

Summary

- The definitions and functionalities of constructors and destructors of classes were discussed.

- The following keywords and their use were discussed:

Keyword	Use
`explicit`	Prevents the compiler from implicit casting when one-argument constructor is called.
`static`	Defines a static data member (common among all objects of a given class).
`friend`	Gives access to private data members of a class for a non-member function.
`this`	A default pointer pointing to the object that is making a function call.