C++ Programming Bootcamp 2

COSC2802

TOPIC 7 Objects and Classes I

Workshop Overview

- Classes!
- Separate Files
- Access, Overloading, Static Data Members and Functions

Class Basics C++

Can anyone tell me what a class is?

What is a class? Why do we care?

A class is a collection of functions and variables that are related to some concept.

We use classes everywhere:

- std::string, std::vector, std::cout, std::endl.
- Most things from the STL are objects/classes.
- std::string / std::vector are interfaces to basic datatypes: C-Style Arrays

The main purposes of using classes are:

1. **Abstracting functionality** (like functions); allows you to focus on high level logic rather than implementation details.
2. **Encapsulating code**; makes it easy to find implementations in very large projects.

They provide a more consistent and user-friendly way to interact with data.

What is a class? Why do we care?

A class is a collection of functions and variables that are related to some concept.

Restaurant

Data

- Name
- Rating
- Year opened

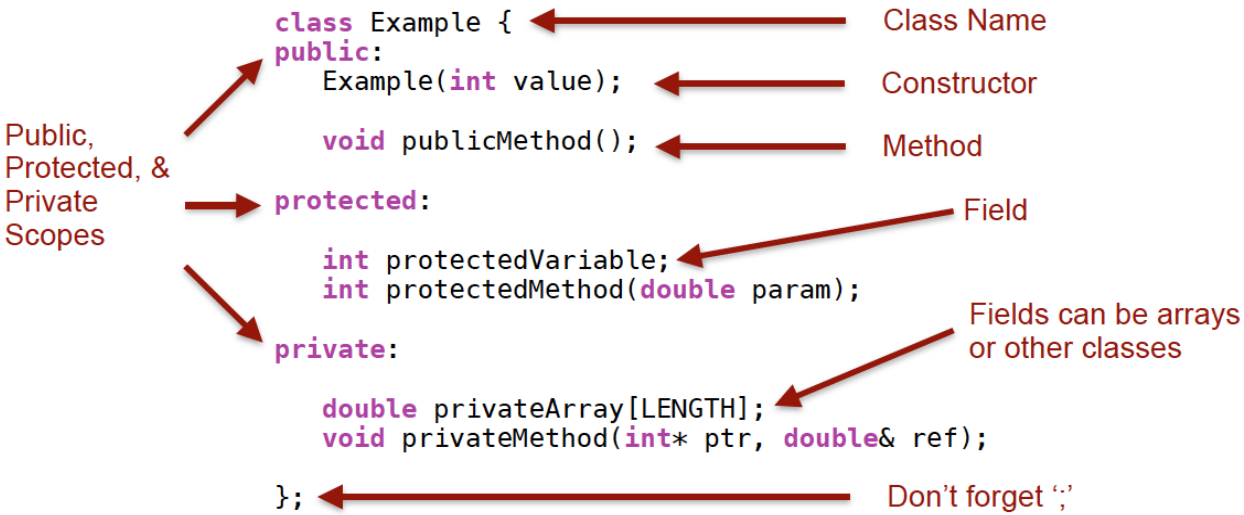
Functions

- Get details
- Submit review

```
#ifndef RESTAURANT_H
#define RESTAURANT_H

class Restaurant
{
public: Restaurant(std::string name = "NoName", int rating = -1);
    void print();

private:
    std::string name;
    int rating;
};

#endif
```

```
#include "Restaurant.h"

Restaurant::Restaurant(std::string initName, int initRating)
{
    name = initName;
    rating = initRating;
}

void Restaurant::print()
{
    std::cout << "Restaurant " << name << " Rating " << rating << "\n";
}
```

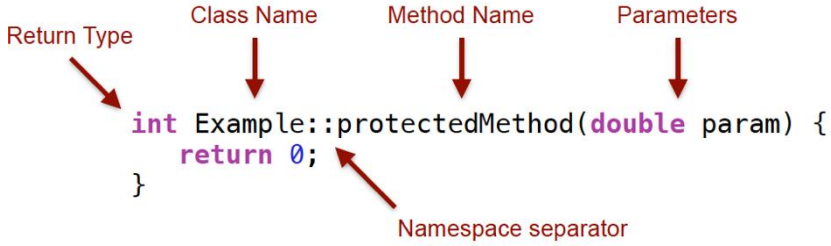
C++ Class Declaration



- Public
- Protected: revisit later I course (zybooks: Access by Members of Derived Class)
- Private

Class Method Definitions

- C++ Class method definitions provide the implementation of each method
  - Definitions provided individually
  - Scope is not relevant to the definition
  - The Class name creates a namespace!



NOTE: You can put method definitions within the class declaration (.h file), but you should only do this for very short sections of code to keep readability.

```
class MyClass {
   public:
      void Fct1();
   private:
      int numA;
};

void MyClass::Fct1() {
   numA = 0;
}
```

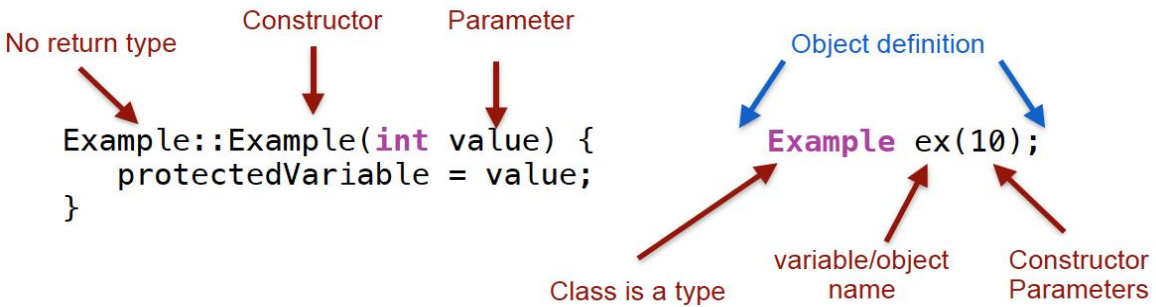
```
class MyClass {
   public:
      void Fct1() {
         numA = 0;
      }
   private:
      int numA;
};
```

Class Initialisation - Constructors

When a class is instantiated into an object; the class's constructor is called.

The constructor is responsible for initializing all variables and data stored by the c

This can be done either by parameters passed into the constructor or setting the variables to known default values.



Accessing Class Members

- Class members (variables and methods) are accessed using dot '.' Syntax

```
Example ex(10);
ex.publicMethod();
```

- Class members can only be accessed from the correct scope
  - Public members are always accessible
  - Private members are only accessible only from within the class
  - Protected members can be accessed from this class and all children (more later)

- For pointers to object, arrow syntax '->' is a shortcut for dereferencing

```
Example* ptrEx = &ex;
(*ex).publicMethod();
ex->publicMethod();
```

Return to this later

Program Layout

Separate Files

Declaration vs Definition vs Initialisation **

- Declaration
  - Introduce a name (variable, class, function) into a scope
  - Fully specify all associate type information
- Definition
  - Fully specify (or describe) the name/entity
  - All definitions are declarations, but not vice versa
- Initialisation
  - Assign a value to a variable for the first time

Separate Files for Classes

Zybooks (Objects and Classes chapter)

Programmers typically put all code for a class into two files, separate from other code.

- **ClassName.h** contains the class definition, including data members and member function declarations.
- **ClassName.cpp** contains member function definitions.

A file that uses the class, such as a main file or ClassName.cpp, must include ClassName.h.

The .h file's contents are sufficient to allow compilation, as long as the corresponding .cpp file is eventually compiled into the program too.

Separate Files for Classes

Good practice for .cpp and .h files:

- Sometimes multiple small related classes are grouped into a single file to avoid a proliferation of files.
- For typical classes, good practice is to create a unique .cpp and .h file for each class

NOTE: Software Engineering Concept:

- Separating Class *Interface* (what it offers) from *Implementation* (how it does it)

Example | Separate Files

File: Restaurant.h (interface)

```
#ifndef RESTAURANT_H
#define RESTAURANT_H

class Restaurant
{
public: Restaurant(std::string name = "NoName", int rating = -1);
    void print();

private:
    std::string name;
    int rating;
};

#endif
```

File: Restaurant.cpp (implementation)

```
#include "Restaurant.h"

Restaurant::Restaurant(std::string initName, int initRating)
{
    name = initName;
    rating = initRating;
}

void Restaurant::print()
{
    std::cout << "Restaurant " << name << " Rating " << rating << "\n";
}
```

Use of Header Guard

File: main.cpp

```
#include <iostream>
#include "Restaurant.h"

int main()
{
    Restaurant foodPlace;              // Calls default constructor
    Restaurant coffeePlace("Joes", 5); // Calls another constructor
    foodPlace.print();
    coffeePlace.print();
    return EXIT_SUCCESS;
}
```

Access, Overloading, Static Data Members and Functions

Overloading | Function/Method

- ***Overloading*** is where the same name is used for multiple functions/methods
  - The overloaded functions/methods must have different types**
    - In C++ this means the parameters must be different
    - Be careful about auto-type casting (see later lecture)
  - The overload method that is called is determined by the method's type

```
int printCard(Card& card);
int printCard(Card* card);
int printCard(Colour colour, int number);
```

- You c*annot* overload by changing the return type, eg. this will *not* work

```
int getColour();
std::String getColour();
```

Auto-casting or implicit type conversion (e.g. *int* to *double*) can cause incorrect / unpredictable behaviour e.g. ambiguity

Constructor Overloading

- Multiple constructors can be created on a class
  - Strictly this “overloads” the constructor operator
  - We’ve already seen this through default, “normal” and copy constructors!
  - Of course other constructors can be defined

```
Card();
Card(int colour, int number);
Card(std::string colour, int number);
Card(Card& other);
```

The compiler does automatically generate a default constructor (*Card()*)

BUT: If any constructor is defined, compiler will not do this

- So, it is important to define a default constructor just in case!

Constructor initializer list

A ***constructor initializer list*** is an alternative approach for initializing data members in a constructor, coming after a colon and consisting of a comma-separated list of variableName(initValue) items.

You can initialise default values in two ways:

1. Just by setting them using normal statements.
2. By using a constructor initializer list.

```
class ExampleClass {
    public:
        ExampleClass();
    private:
        int field1;
        int field2;
};
```

```
ExampleClass::ExampleClass() {
    field1 = 1;
    field2 = 2;
}
```

```
ExampleClass::ExampleClass() : field1(1), field2(2) {}
```

Why we use namespaces:

Naming conflicts!

It is quite likely that variables from different classes will use similar naming conventions, and so names will often be duplicated. This can cause significant issues in compilation, as all names (variables, functions, classes, etc.) must be unique.

main.cpp

```
#include "auditorium.h"
#include "airplane.h"

int main() {
    auditorium::Seat concertSeat;
    airplane::Seat flightSeat;

    // ...

    return 0;
};
```

auditorium.h

```
namespace auditorium {
    class Seat {
        ...
    };
}
```

airplane.h

```
namespace airplane {
    class Seat {
        ...
    };
}
```

We use **namespaces** to dictate the scope for variables/classes/functions to resolve these naming conflicts or avoid them.

THIS IS WHY USING NAMESPACE STD IS BAD! It defeats the point of having namespaces at all!

Static Data Members and Functions

Static keyword:

- Indicates variable only allocated memory once (value can change)

Static data member:

- data member of **class** instead of member of each **class object**
- Independent of any class object and can be accessed **without creating class object**

Declaration and Definition:

- Declared **inside** class definition. Defined **outside** declaration

Accessed:

- Within class by name. Outside class using the scope resolution operator

Static Data Member Example

The Variable static_field is shared across all instantiations of ExampleClass.

```
class ExampleClass {
    public:
        ExampleClass();
        void print_static();
        void set_static(int);
    private:
        static int static_field;
};

ExampleClass::ExampleClass() {}
void ExampleClass::print_static() { std::cout << static_field << std::endl;}
void ExampleClass::set_static(int i) { static_field = i; }
int ExampleClass::static_field = 1;
```

```
ExampleClass ex1;
ExampleClass ex2;

ex1.print_static();
ex2.print_static();

ex1.set_static(100);

ex1.print_static();
ex2.print_static();
```

If we run this:

```
$ ./ex
1
1
100
100
```