COMP 116

Chapter 3: Classes and Objects

Department of Computer Science and Engineering
Kathmandu University

Rajani Chulyadyo, PhD

Contents

- C++ classes and objects
  - Example
- Private member functions
- Static data members
- Static member functions

Procedural programming

In “procedural” programming, problems are “decomposed” into smaller units of repeatable activities called *procedures*.

Disadvantages:

- Importance is given to the operation on data rather than the data.
- No information hiding: data is exposed to whole program.
- Difficult to relate with real-world objects
- Difficult to manage large programs

Object-Oriented Programming (OOP)

- The fundamental idea:
  - Combine into a single unit both data (attributes) and the functions that operate on that data (functionality/behavior).
- In this paradigm, a computer program is built with a collection reusable components called **objects**.
- Related piece of information and behaviours are organized together into a template called a **class**.
- A class is thus a description of a number of similar objects.
- **An object is an instance of a class.**

Data encapsulation and data hiding

Data encapsulation:

- Combine into a single unit (called object) both **data (**aka **attributes** or **member variables** or **data members)** and the **functions** that operate on that data (aka **member functions** or **methods**).
- A **class** is a description of a number of similar objects.

Data hiding:

- Restricting direct access to the data (so that it is safe from accidental alteration).
- In C++, access specifiers/modifiers determine the accessibility of data members.

C++ class

Example:

Write a program that calculates the Euclidean distance between two points (x_1, y_1) and (x_2, y_2). Distance = $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

C++ class

Example:

Write a program that calculates the Euclidean distance between two points (x_1, y_1) and (x_2, y_2). Distance = $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

Solution:

- What can be our objects here? Two points (x_1, y_1) and (x_2, y_2)

C++ class

Example:

Write a program that calculates the Euclidean distance between two points (x_1, y_1) and (x_2, y_2). Distance = $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

Solution:

- What can be our objects here? Two points (x_1, y_1) and (x_2, y_2)
- What are common between them? Both have x- and y-coordinates.

C++ class

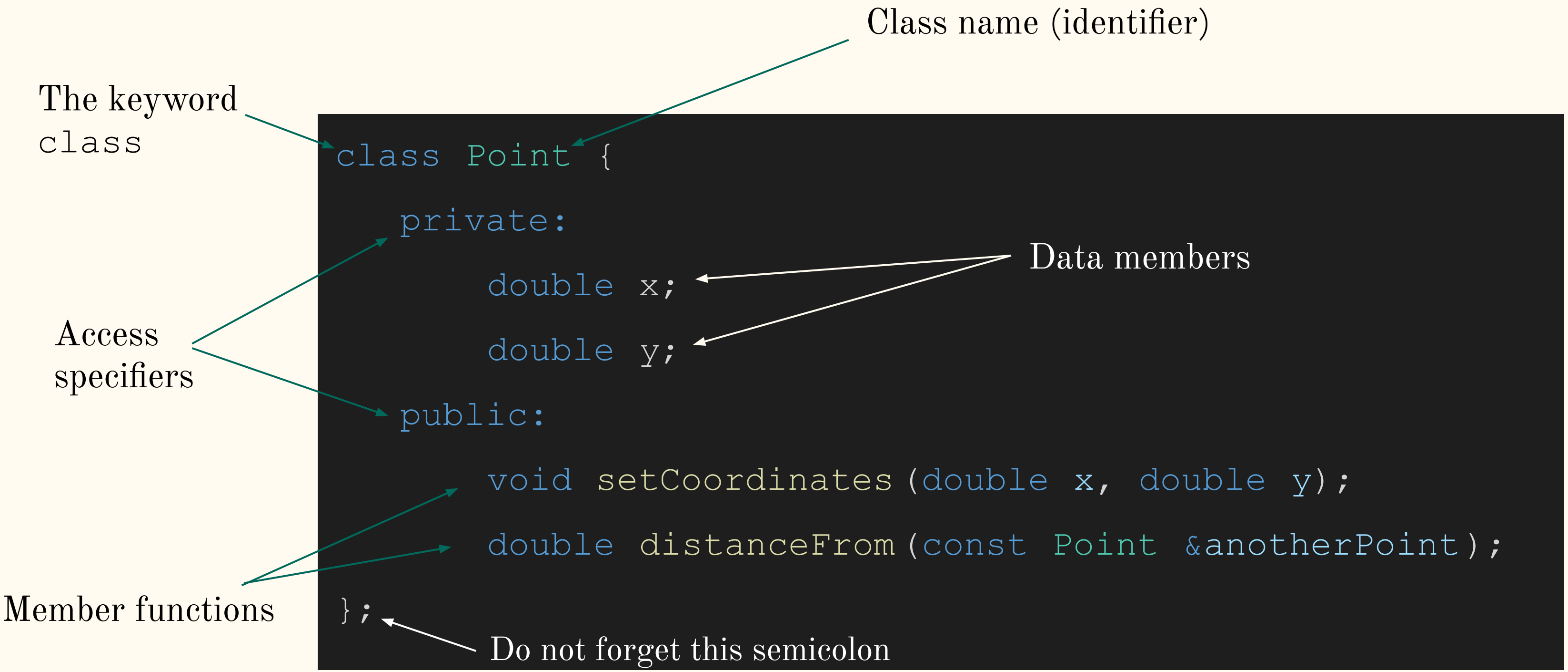
Example:

Write a program that calculates the Euclidean distance between two points (x_1, y_1) and (x_2, y_2). Distance = $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

Solution:

- What can be our objects here? Two points (x_1, y_1) and (x_2, y_2)
- What are common between them? Both have x- and y-coordinates.
- What operations are needed?
  - Initialize coordinates
  - Calculate distance from one point to another.

The `Point` Class: Declaration



```
class Point {
   private:
       double x;
       double y;
   public:
       void setCoordinates(double x, double y);
       double distanceFrom(const Point &anotherPoint);
};
```

Class name (identifier)

The keyword `class`

Data members

Access specifiers

Member functions

Do not forget this semicolon

A **class declaration** describes the type and the scope of its members.

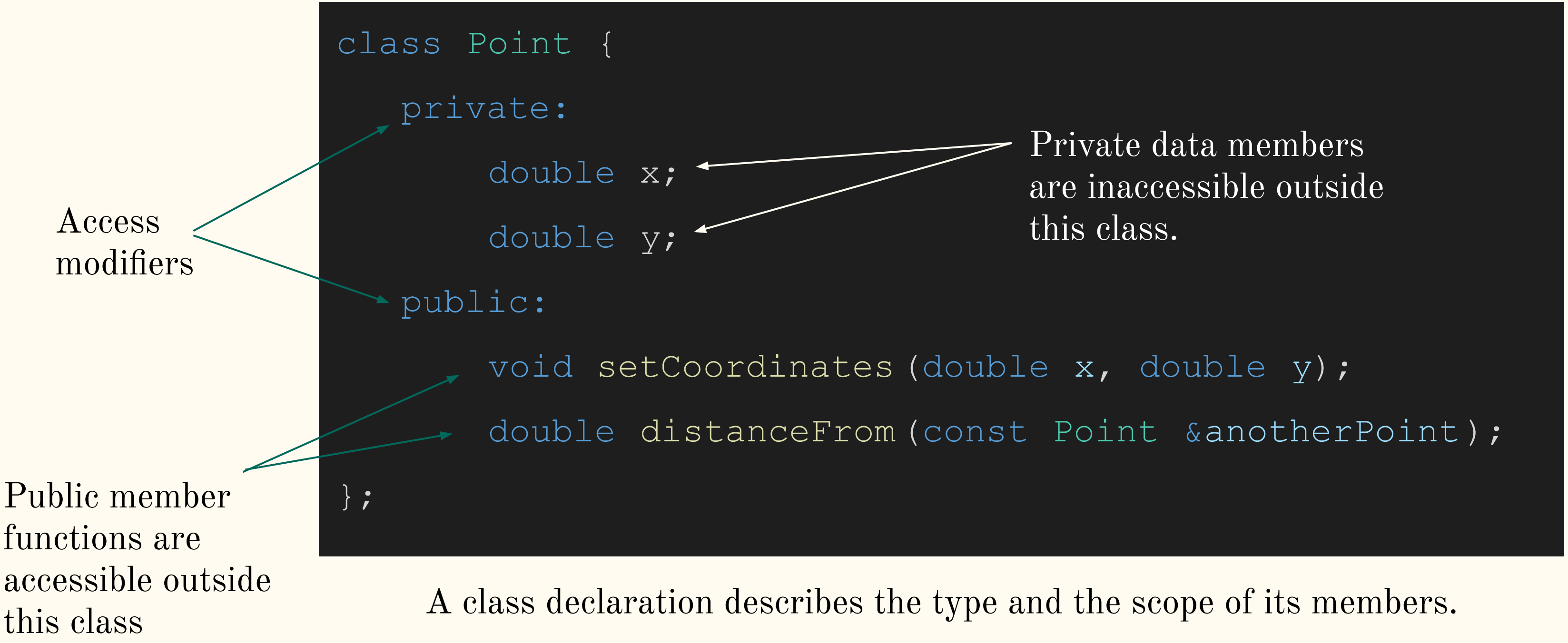
Access specifier

Access specifiers (aka **access modifiers**) define how the members of a class can be accessed.

In C++, there are 3 types of access specifiers:

1. **`public`** - members are accessible from outside the class.
2. **`private`** - members can be accessed only by the methods inside the class and are inaccessible outside the class.
3. **`protected`** - members are inaccessible outside the class but they can be accessed by any subclass (derived class) of that class. (Will be covered in Chapter 6.)

The `Point` Class: Declaration



```
class Point {
  private:
    double x;
    double y;
  public:
    void setCoordinates(double x, double y);
    double distanceFrom(const Point &anotherPoint);
};
```

Access modifiers

Private data members are inaccessible outside this class.

Public member functions are accessible outside this class

A class declaration describes the type and the scope of its members.

Creating an object

An **object** is an **instance of a class.**

p1 and p2 are objects of class Point.

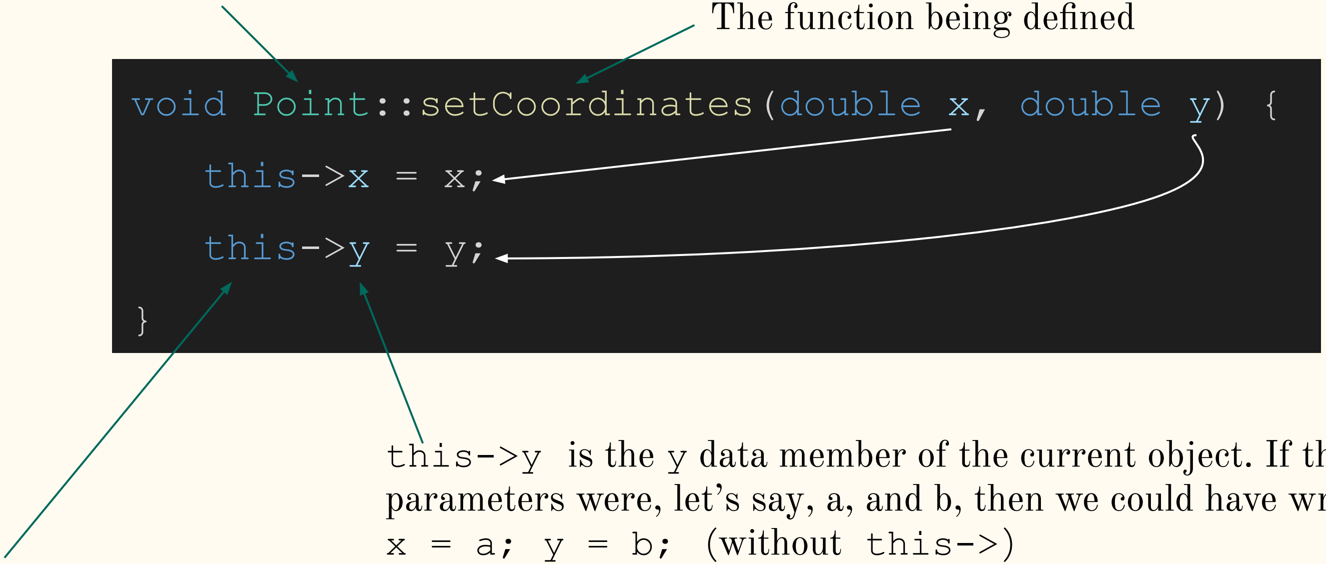
```
Point p1;
p1.setCoordinates(10, 20);


Point p2;
p2.setCoordinates(0, 10);
```

To call a public member function of the object or a public data member, we use a dot (.), e.g., `p1.setCoordinates`

The `Point` Class: Function definition

`Point::` tells the compiler that the following function belongs to the `Point` class.

The function being defined

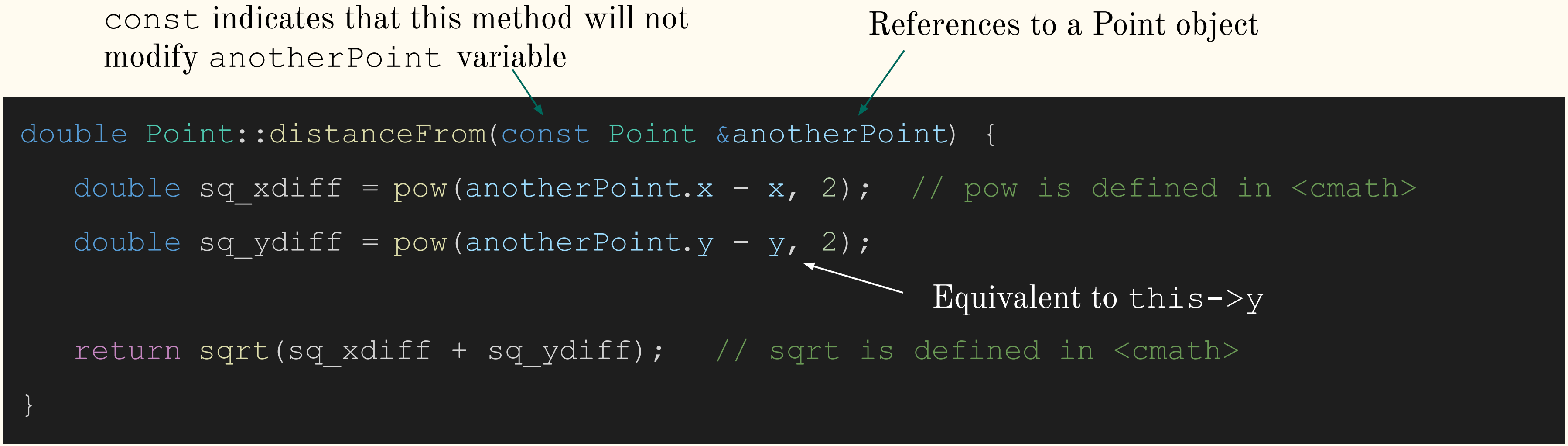


```
void Point::setCoordinates(double x, double y) {
    this->x = x;
    this->y = y;
}
```

`this` is a pointer to the object whose member function is being executed.

`this->y` is the `y` data member of the current object. If the formal parameters were, let's say, a, and b, then we could have written `x = a; y = b;` (without `this->`)
Recall variable scopes.

The Point Class: Function definition



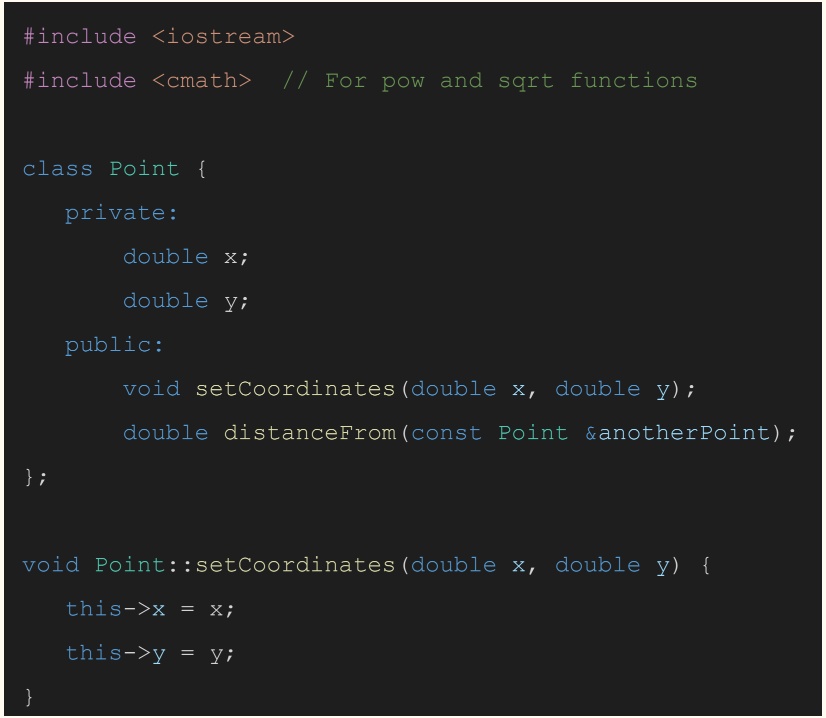
const indicates that this method will not modify anotherPoint variable

References to a Point object

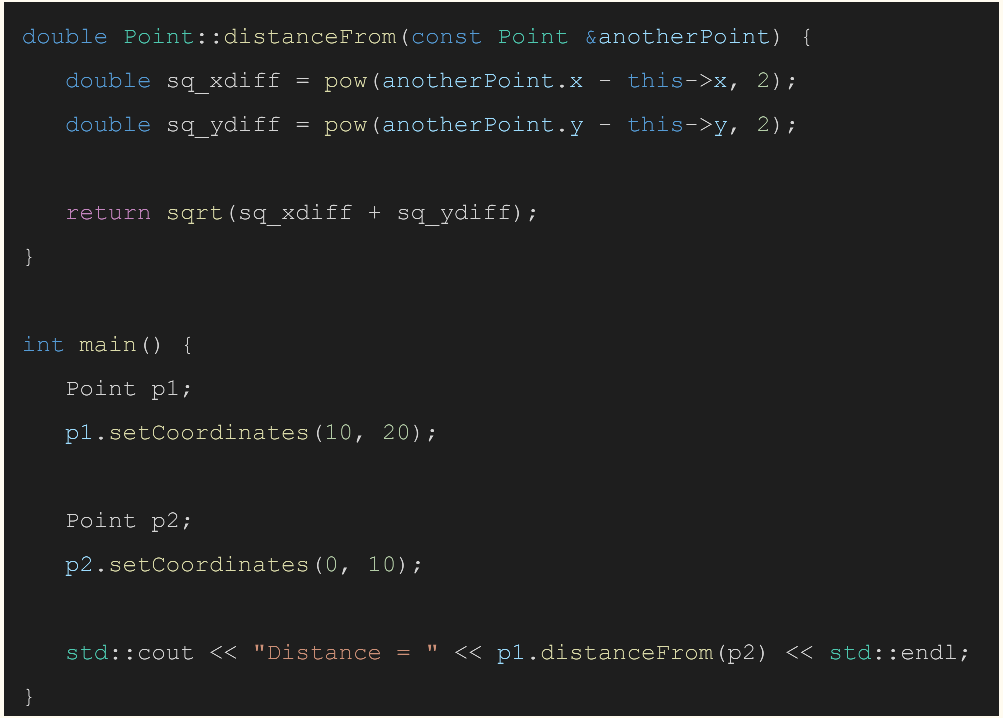
```
double Point::distanceFrom(const Point &anotherPoint) {
   double sq_xdiff = pow(anotherPoint.x - x, 2);  // pow is defined in <cmath>
   double sq_ydiff = pow(anotherPoint.y - y, 2);

   return sqrt(sq_xdiff + sq_ydiff);   // sqrt is defined in <cmath>
}
```

Equivalent to this->y

The complete program



```
#include <iostream>
#include <cmath>  // For pow and sqrt functions

class Point {
   private:
       double x;
       double y;
   public:
       void setCoordinates(double x, double y);
       double distanceFrom(const Point &anotherPoint);
};

void Point::setCoordinates(double x, double y) {
   this->x = x;
   this->y = y;
}
```



```
double Point::distanceFrom(const Point &anotherPoint) {
   double sq_xdiff = pow(anotherPoint.x - this->x, 2);
   double sq_ydiff = pow(anotherPoint.y - this->y, 2);

   return sqrt(sq_xdiff + sq_ydiff);
}

int main() {
   Point p1;
   p1.setCoordinates(10, 20);

   Point p2;
   p2.setCoordinates(0, 10);

   std::cout << "Distance = " << p1.distanceFrom(p2) << std::endl;
}
```

```
g++ -o PointProgram.out PointProgram.cpp
```

C++ Separate Header and Implementation Files

In practice, we do not keep all our code in a single file, not only for code clarity but also for code reusability, implementation hiding, and compilation time reduction.

Class declarations are kept in a separate file with a .h extension (called a **header file**).

Class function definitions are kept in a file with a .cpp extension (called an implementation file or a **source file**).

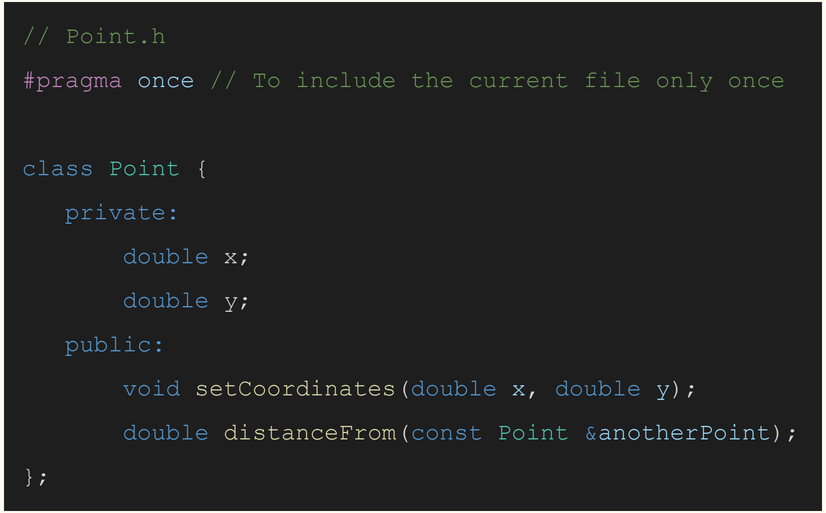
The `main()` function is kept in a file with a .cpp extension (called **client code**).

The Point class example (Contd.)

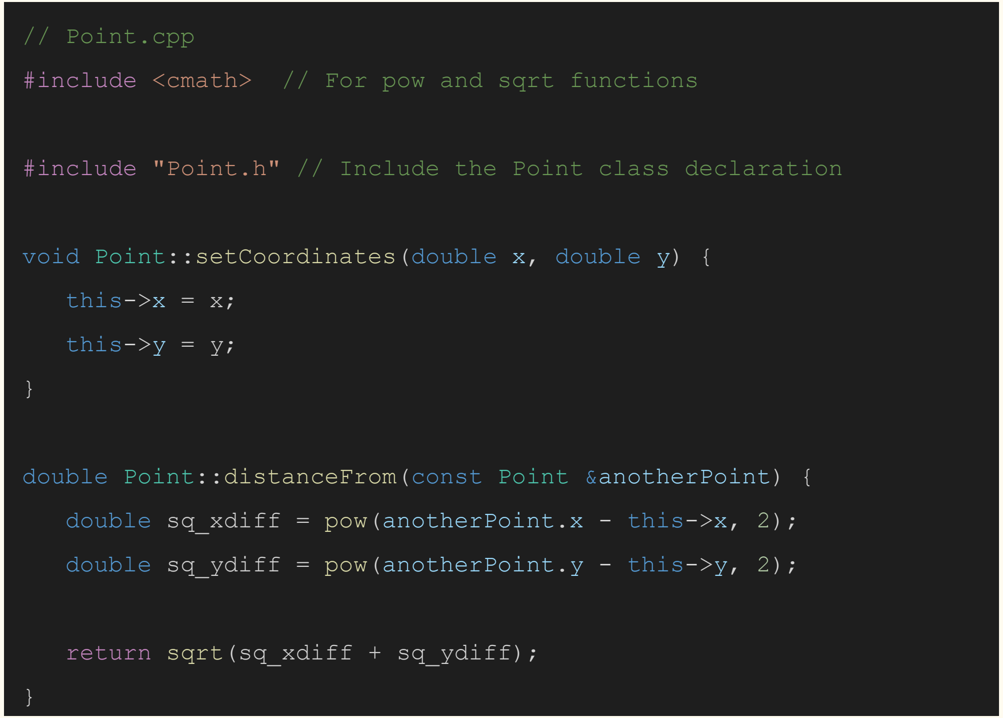
For the `Point` class example, we will create 3 files:

1. Point.h that contains the `Point` class declaration
2. Point.cpp that contains the `Point` class implementation
3. main.cpp that contains the `main` function

The Point class example (Contd.)

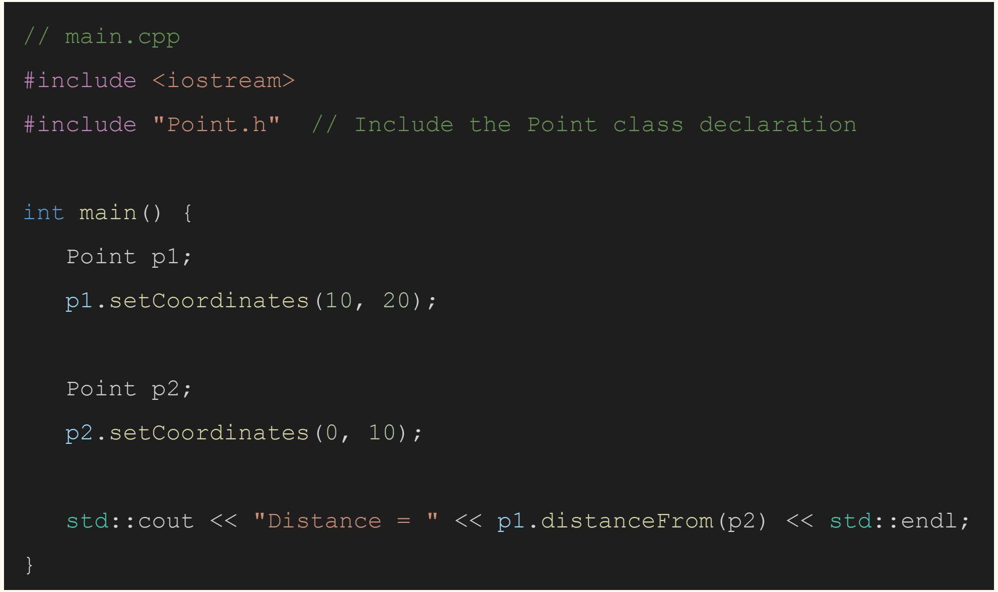


```cpp
// Point.h
#pragma once // To include the current file only once

class Point {
   private:
       double x;
       double y;
   public:
       void setCoordinates(double x, double y);
       double distanceFrom(const Point &anotherPoint);
};
```



```cpp
// Point.cpp
#include <cmath>  // For pow and sqrt functions

#include "Point.h" // Include the Point class declaration

void Point::setCoordinates(double x, double y) {
   this->x = x;
   this->y = y;
}

double Point::distanceFrom(const Point &anotherPoint) {
   double sq_xdiff = pow(anotherPoint.x - this->x, 2);
   double sq_ydiff = pow(anotherPoint.y - this->y, 2);

   return sqrt(sq_xdiff + sq_ydiff);
}
```

The Point class example (Contd.)



```cpp
// main.cpp
#include <iostream>
#include "Point.h"  // Include the Point class declaration

int main() {
    Point p1;
    p1.setCoordinates(10, 20);

    Point p2;
    p2.setCoordinates(0, 10);

    std::cout << "Distance = " << p1.distanceFrom(p2) << std::endl;
}
```

To build the program:

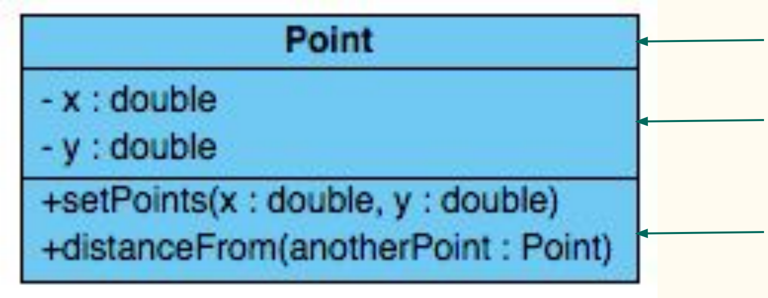
`g++ -o Point.out` **`main.cpp Point.cpp`**

Abstraction

- Providing only essential information to the outside world and hiding their background details.
- Its main goal is to handle complexity by hiding unnecessary details from the user.
- Abstraction separates code into **interface** and **implementation**.
- Interface should be independent of the implementation so that if the underlying implementation is changed, the interface would remain intact.
- In C++, we can have interface in header files as class declarations, and implementation in source files that contain class function definitions.

UML class diagram

A class diagram gives an overview of a software application by presenting the classes, and relations between them.


Point
- x : double
- y : double
+setPoints(x : double, y : double)
+distanceFrom(anotherPoint : Point)

Class name

Attributes / member data

Operations / member functions

-, +, # are used to indicate the access specifier of the member. that the member is private, public or protected.

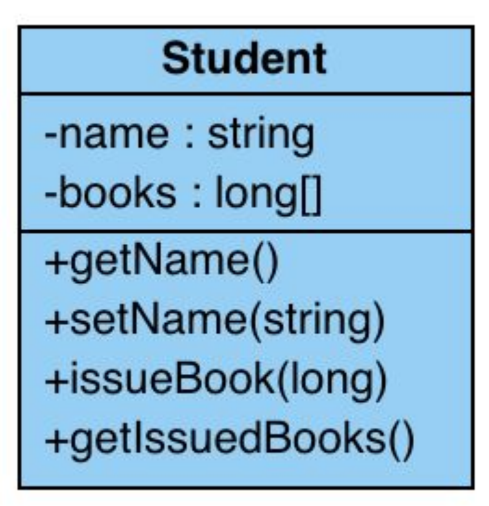
- for private
+ for public.
# for protected.

More on class diagrams in later chapters

Lab 2 exercise

Question 1

Implement a class called Student, shown in the class diagram below:



Student

-name : string
-books : long[]

+getName()
+setName(string)
+issueBook(long)
+getIssuedBooks()

Data members:

- Name of the student
- Array of the ID of the books the student has borrowed

Member functions:

- getName() returns the name of the student
- setName(string) sets the name of the student
- issueBook(long) adds the ID of the book borrowed by the student to the array of borrowed books' ID
- getIssuedBooks() returns the IDs of the borrowed books

Lab 2 exercise

Question 2

Define an enum called Grade, with the following values: A, A-, B+, B, B-, C+, C, C-, D, and F.

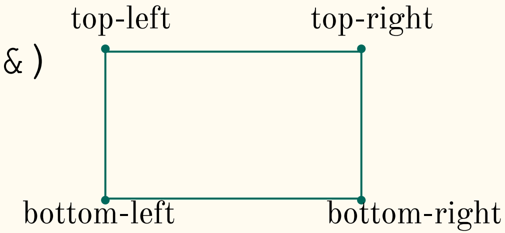
Modify Student class (defined in previous question) to store the grade of the student by adding a data member of type Grade and its getter and setter.

Lab 2 exercise

Question 3

A rectangle can be defined by two points (top-left and bottom-right or top-right and bottom-left). Implement a class called `Rectangle` using the `Point` class we saw during the lecture. The `Rectangle` class must have the following methods:

1. `void setPoints(const Point &, const Point &)`
2. `void getDimensions(double &, double &)`
3. `double perimeter()`



Write the `main()` function to check if your implementation works correctly.

(20 mins)

Private member function

A member function may be private, public, or protected.

A private member function cannot be accessed from outside the class.

A private member function

```
class Point {
   private:
       double x, y;
       double distanceFrom(double x, double y);
   public:
       void setCoordinates(double x, double y);
       double distanceFrom(const Point &anotherPoint);
};
```

Private member function

No difference in the definition of a private member function.

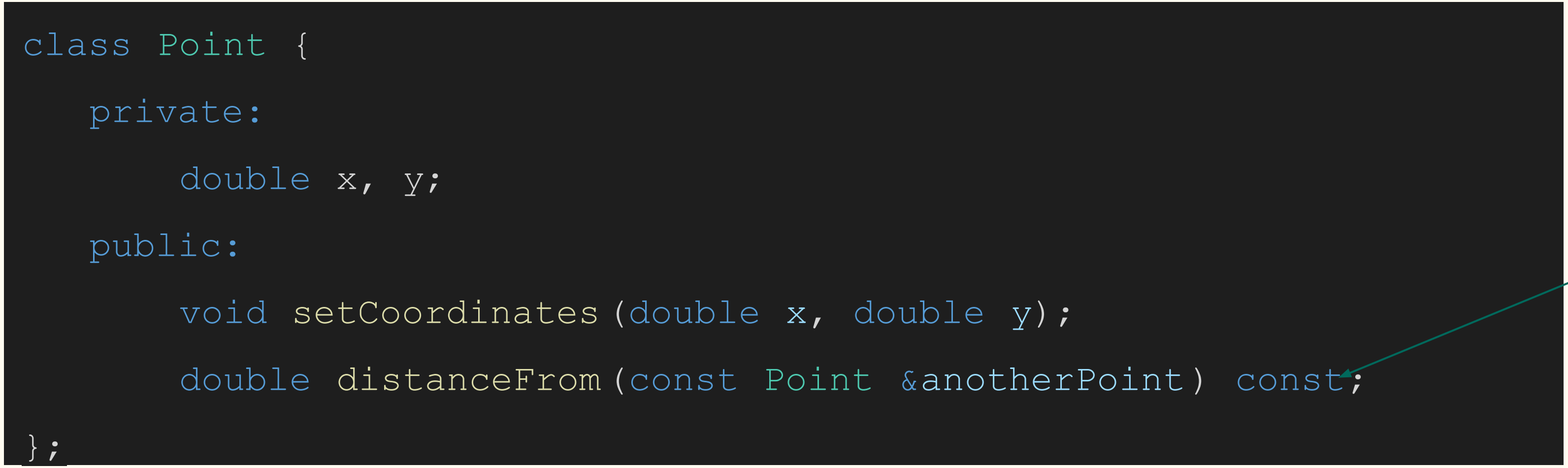
```
double Point::distanceFrom(double x, double y) {
   double sq_xdiff = pow(x - this->x, 2);
   double sq_ydiff = pow(y - this->x, 2);

   return sqrt(sq_xdiff + sq_ydiff);
}


double Point::distanceFrom(const Point &anotherPoint) {
   return distanceFrom(anotherPoint.x, anotherPoint.y);
}
```

A private member function can be called from another member function of the same class but cannot be called from outside the class.

const member function

A const member function guarantees it will not modify the object or call any non-const member functions (as they may modify the object).

A const member function contains the keyword `const` in its declaration.



```
class Point {
   private:
       double x, y;
   public:
       void setCoordinates(double x, double y);
       double distanceFrom(const Point &anotherPoint) const;
};
```

The `const` keyword

Static data members

When we instantiate a class object, each object gets its own copy of all normal (non-static) data members whereas **static data members are shared by all objects of the class**.

Static data members belong to the class itself.

Static members exist even if no objects of the class have been instantiated.

They are created when the program starts, and destroyed when the program ends.

They can be accessed directly using the class name and the scope resolution operator (`Classname::static_member_variable`).

Static data members

```cpp
#include <iostream>
class StaticDemo {
   public:
       static int var;
};
int StaticDemo::var = 0;  // Initialization

int main() {
   std::cout << "Static member data = " << StaticDemo::var++ << std::endl;  // 0
   StaticDemo::var = 10;
   std::cout << "Static member data = " << StaticDemo::var << std::endl;    // 10
}
```

Static member functions

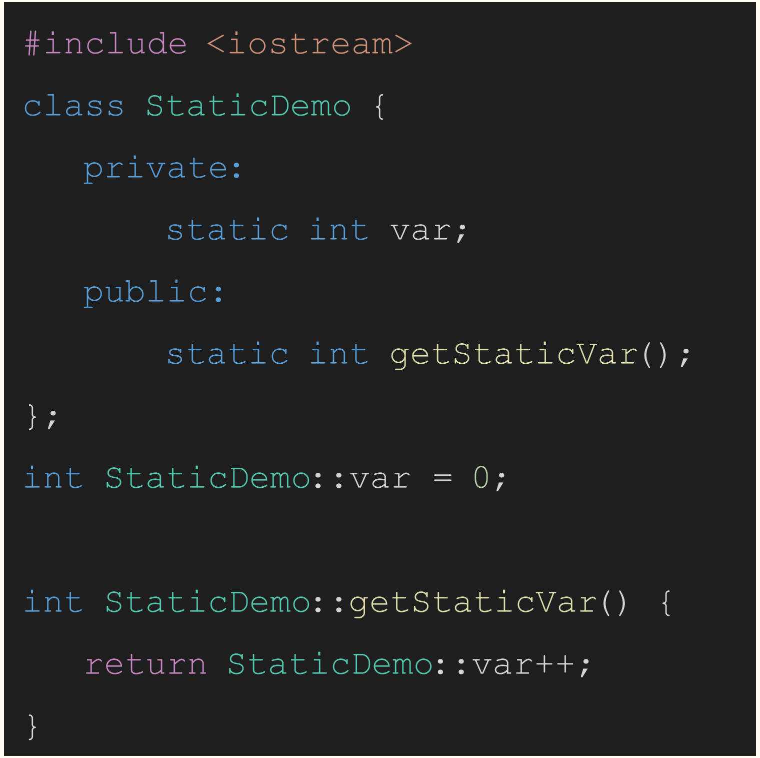
Like static member variables, static member functions are not attached to any particular object.

They can directly access other static members (variables or functions), but not non-static members.

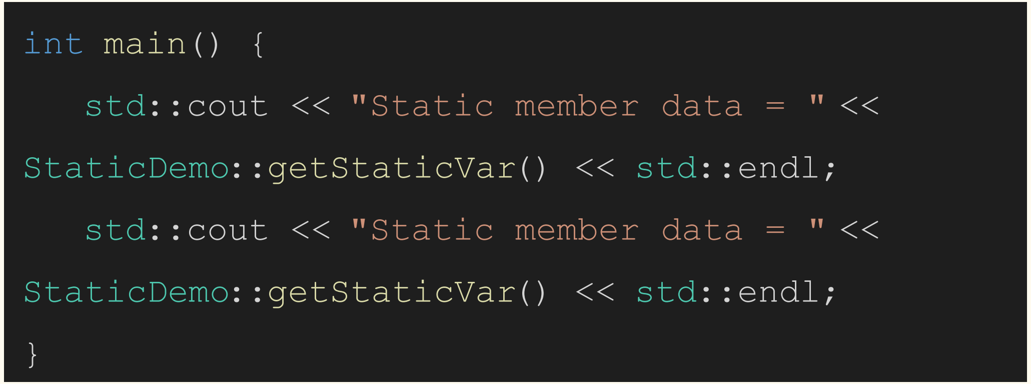
They have no `this` pointer.

They can be called directly by using the class name and the scope resolution operator.

Static data members



```
#include <iostream>
class StaticDemo {
   private:
       static int var;
   public:
       static int getStaticVar();
};
int StaticDemo::var = 0;

int StaticDemo::getStaticVar() {
   return StaticDemo::var++;
}
```



```
int main() {
   std::cout << "Static member data = " <<
StaticDemo::getStaticVar() << std::endl;
   std::cout << "Static member data = " <<
StaticDemo::getStaticVar() << std::endl;
}
```

Resources

1. https://en.cppreference.com/w/cpp/
2. http://cplusplus.com/doc/tutorial/
3. https://www.learncpp.com/
4. https://www.edureka.co/blog/namespace-in-cpp/