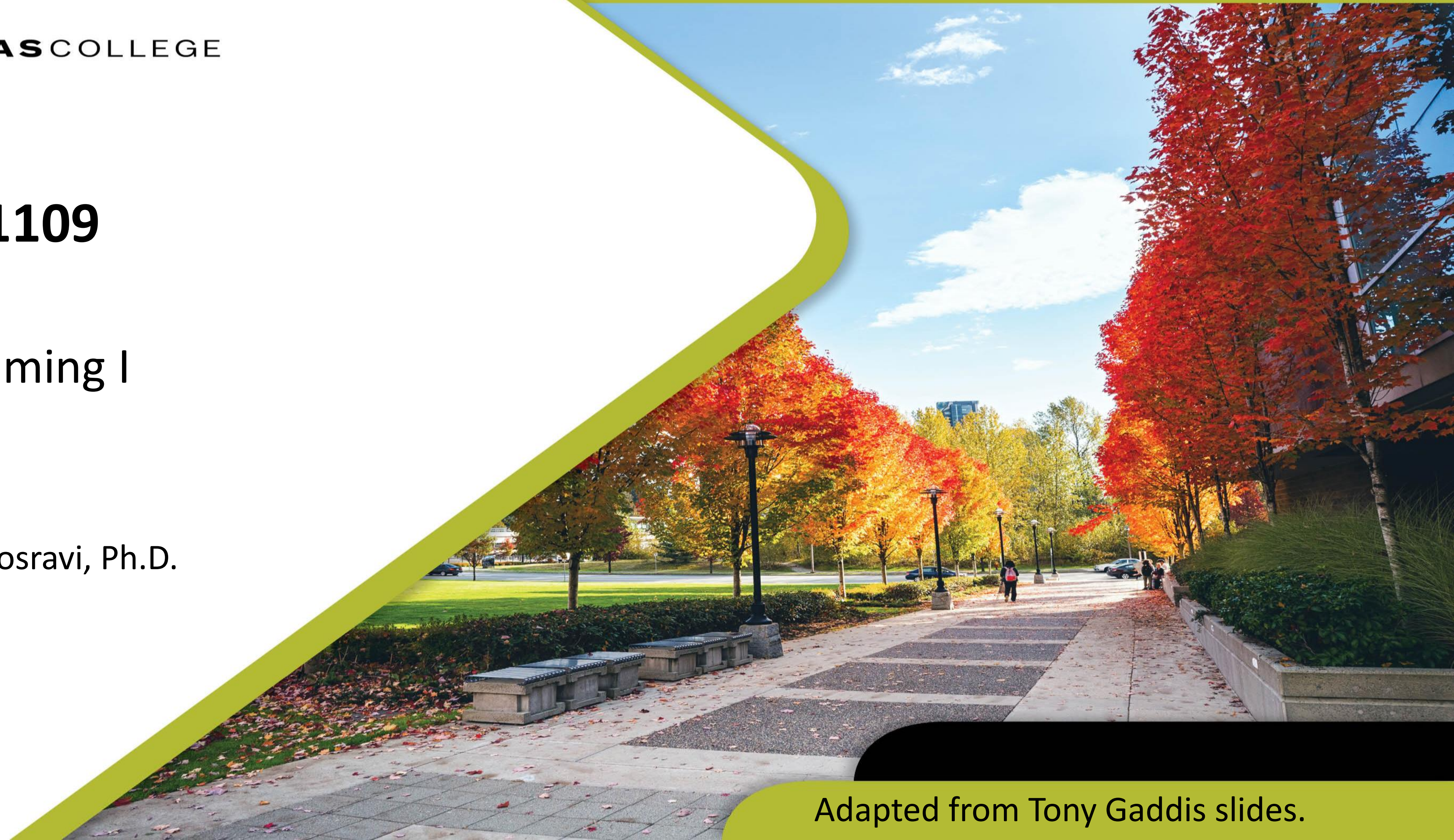
CMPT 1109

Programming I

Shahriar Khosravi, Ph.D.

Lecture 11



Adapted from Tony Gaddis slides.

Plan for Today

- Object-Oriented Programming
- Structures in C++
- Objects and Classes

DOUGLAS COLLEGE

Object-Oriented Programming

Object-Oriented Programming (OOP)

- Object-Oriented Programming was developed because limitations were discovered in earlier approaches to programming including structured (procedural) programming.
- To appreciate what OOP does, we need to understand these limitations and how they arose from procedural programming languages .
- C, Fortran, and similar languages are considered procedural languages – each statement in the language tells the computer to do something.
- A program in a procedural language is a list of instructions.
- For very small programs, no other organizing principle (often called a paradigm) is needed.

Structured Programming

- When programs become larger, a single list of instructions becomes too complicated to follow.
- For this reason, functions were adopted as a way to make programs more comprehensible.
- A procedural program is divided into functions, and each function has a clearly defined purpose, and a clearly defined interface to the other functions.
- A number of functions may also be grouped together (e.g. in a header file) into a larger entity called a “module”.
- Dividing a program into a number of functions and modules is called **structured programming**.

Structured Programming

- No matter how well the structured programming approach is implemented, large programs become excessively complex.
- Why? There are two main reasons.
- First, functions have unrestricted access to global data.
- Second, unrelated functions and data (i.e. the basis of procedural programming), provide a poor model of the real world.
- Example: a change made in a global data item may necessitate rewriting all the functions that access that item.

Real-World Modelling

- In the real world, we deal with objects such as cars, airplanes, etc.
- Real-world objects have **attributes** and **behavior**.
- Examples of attributes (or characteristics) are color, horsepower, model year, etc.
- Attributes of objects in the real world are equivalent of data in a program: they have specific values such as blue, 182hp, or 2020.
- Behavior is something a real-world object does in response to some stimulus (e.g. if you apply the brakes, the speed of the car decreases).
- Behavior is like a function: you call a function to do something and it does it.
- So, neither data nor functions, by themselves, model real-world objects effectively.

Object-Oriented Programming (OOP)

- The fundamental idea behind object-oriented languages such as C++ is *to combine attributes and behaviour into a single unit*.
- Such a unit is called an **<u>object</u>**.
- In C++, an object's functions are called **<u>member functions</u>**, and they typically provide the only way to access the object's data.
- If you want to read a **<u>hidden</u>** data item in an object, you will call a member function in the object, which will access the data and return the value to you.
- You can modify access privileges such that the object's data is *hidden*, so it is safe from accidental alteration (i.e. *data hiding* and *encapsulation*).

DOUGLAS COLLEGE

Structures in C/C++

Structures in C++

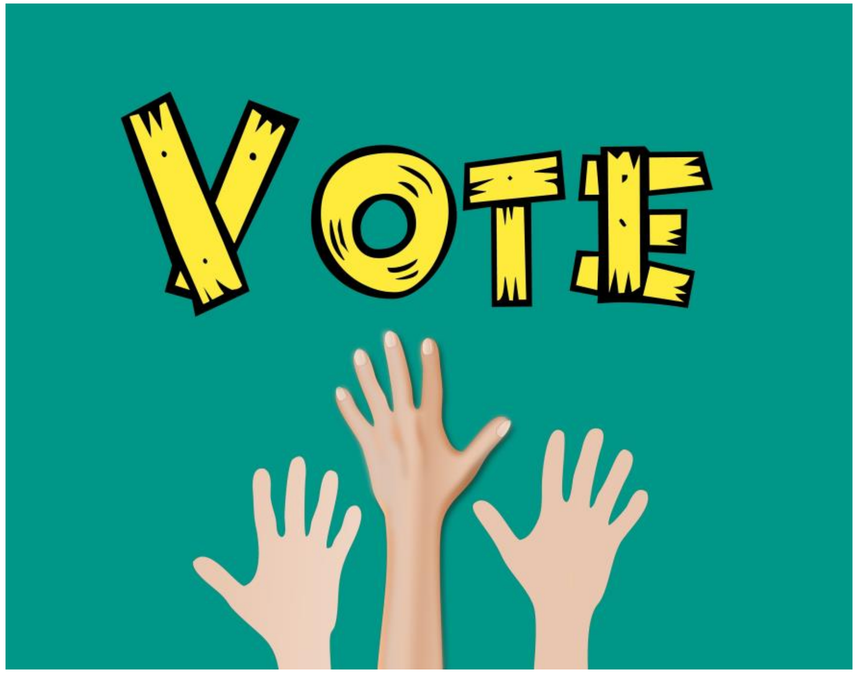
- A **`structure`** is a collection of simple variables.
- The variables in a structure can be of different types: some can be **`int`**, some can be **`float`**, and so on (unlike arrays).
- In C programming, structures are often introduced as an advanced feature.
- In C++ programming, structures are one of the two important building blocks in the understanding of **objects** and **classes**.
- In fact, the syntax of a **`structure`** is almost identical to that of a **`class`**.

Poll 1 (Extra Credit)

A `structure` definition by itself sets aside memory in RAM for the structure variables.

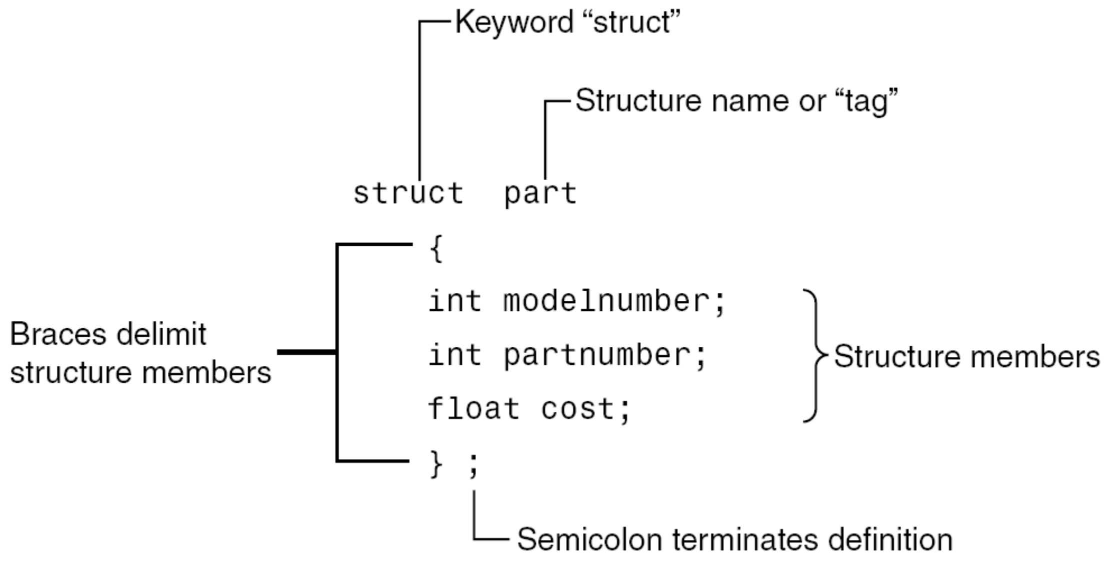
a) True
b) False
c) Not sure!

Please use the "Poll" window to participate for extra credit! One answer only please!



C++ Syntax for Structures

- The keyword **`struct`** introduces the definition.
- Next comes the **structure name** or **tag**.
- The structure members are enclosed in braces.
- A semicolon terminates the closing brace.

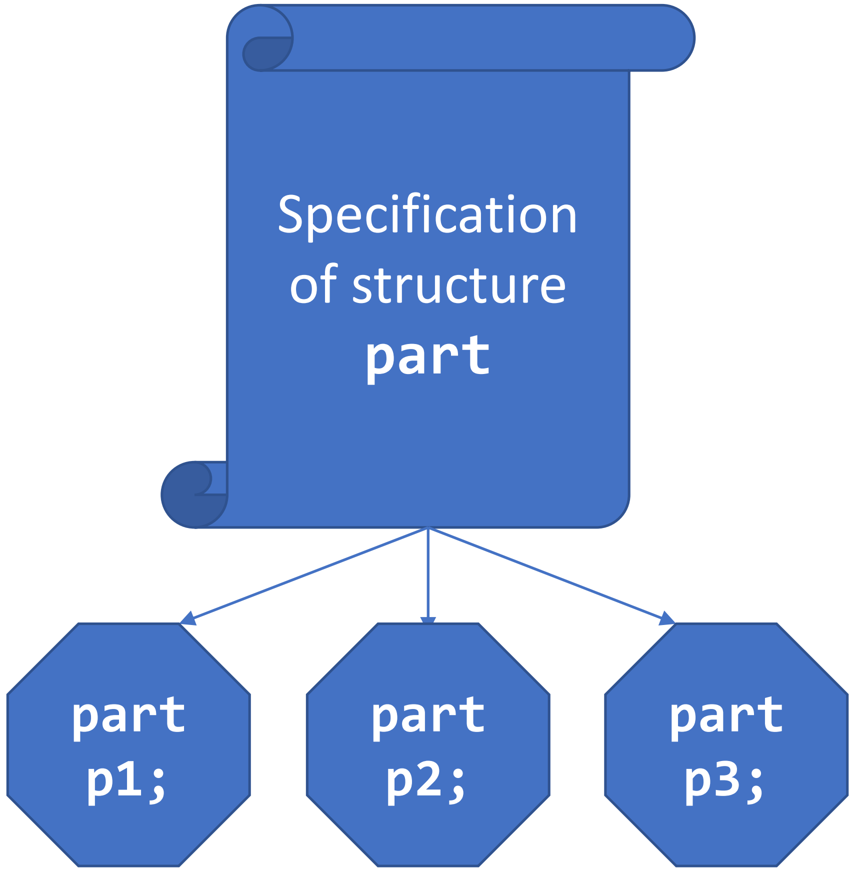


```
struct   part
  {
  int modelnumber;
  int partnumber;
  float cost;
  } ;
```

Keyword "struct"
Structure name or "tag"
Braces delimit structure members
Structure members
Semicolon terminates definition

Defining a New `struct` Variable

- Following the definition of the **`structure part`**, we can instantiate a new variable **`p1`** of the type **`part`**.
- We can think of structures as definitions for new data types.
- **`int var1;`**
- **`part part1;`**



Accessing Structure Members

- Once a structure variable has been defined, its members can be accessed using the **dot operator** (i.e. the **member access operator**).

```
part part1;
part1.modelnumber = 6244;
```

- Structure members are treated just like other variables.=
- As a result, we can use the **assignment operator (=)** to assign a value to the **`modelnumber`** member of **`part1`**.

Structure Example

- The **structure part** has two **int** and one **float** variables.

- Inside the **main()** function, part1 is a variable of type **part**.

```
#include <iostream>
using namespace std;

struct part //declare a structure
{
    int modelnumber; //model number
    int partnumber;  //part number
    float cost; //cost of part
};

int main()
{
    //define a structure variable
    part part1;

    //give values to structure members
    part1.modelnumber = 6244;
    part1.partnumber = 373;
    part1.cost = 217.55F;

    //display structure members
    cout << "Model " << part1.modelnumber;
    cout << ", part " << part1.partnumber;
    cout << ", costs $" << part1.cost << endl;

    return 0;
}
```

Initializing Structure Members

- This example shows how to initialize **`structure`** members.
- The members are initialized in the same order as they are defined in the **`structure`** definition.

```
#include <iostream>
using namespace std;

struct part
{
    int modelnumber;
    int partnumber;
    float cost;
};

int main()
{
    part part[2];
    part[0] = { 6244, 373, 217.55F };

    cout << "Model " << part[0].modelnumber;
    cout << ", part " << part[0].partnumber;
    cout << ", costs $" << part[0].cost << endl;

    cin >> part[1].modelnumber;
    cin >> part[1].partnumber;
    cin >> part[1].cost;

    cout << "Model " << part[1].modelnumber;
    cout << ", part " << part[1].partnumber;
    cout << ", costs $" << part[1].cost << endl;

    return 0;
}
```

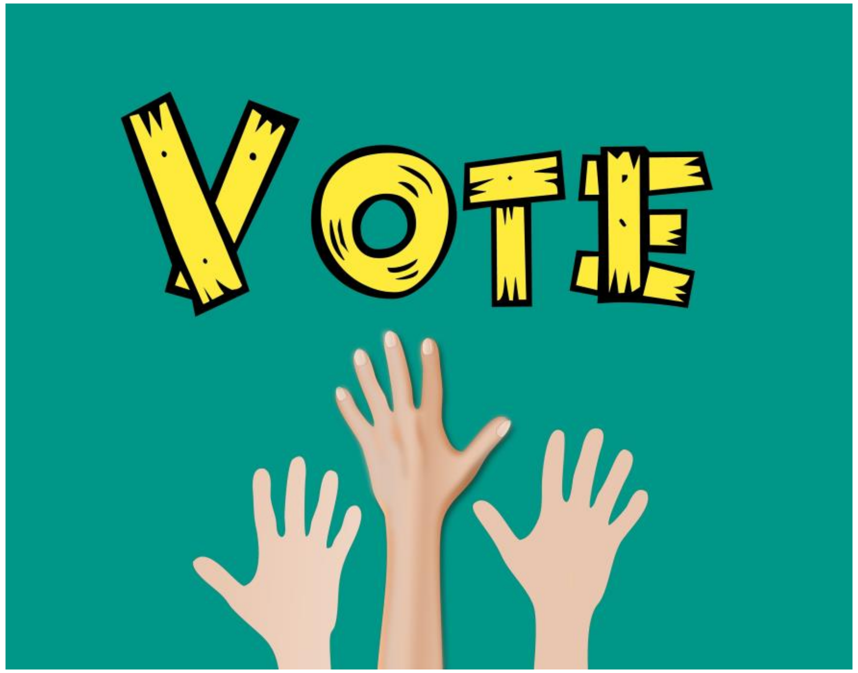
Poll 2 (Extra Credit)

Will the following statements compile successfully?

```
part parts[2];
parts[0] = {21, 23, 21.5F};
parts[1] = parts[0];
```

a) Yes!
b) No!
c) Not sure!

Please use the “Poll” window to participate for extra credit! One answer only please!



Nested Structures

- We can nest structures within other structures.
- This example creates a **structure Distance**.
- A second **structure Room** has two members of type **Distance**.
- But what we still have **not** really defined an object because none of these structures have defined behaviour.

```
#include <iostream>
using namespace std;
struct Distance //English distance
{
    int feet;
    float inches;
};

struct Room //rectangular area
{
    Distance length; //length of rectangle
    Distance width; //width of rectangle
};

int main()
{
    Room dining; //define a room
    dining.length.feet = 13; //assign values to room
    dining.length.inches = 6.5;
    dining.width.feet = 10;
    dining.width.inches = 0.0;
    //convert length & width
    float l = dining.length.feet + dining.length.inches/12;
    float w = dining.width.feet + dining.width.inches/12;
    //find area and display it
    cout << "Dining room area is " << l * w
        << " square feet\n" ;
return 0;
}
```

DOUGLAS COLLEGE

Objects and Classes

Object Example

- In this example, **`struct`** definitions **`Distance`** and **`Room`** have both attributes. (i.e., data) **<u>and</u>** behaviour (i.e., functions)
- Therefore, **<u>instances</u>** of both **`struct`** definitions (e.g., **`dining`**) are referred to as an **<u>object</u>**.
- The **`dining`** object is said to be of type **`Distance`**.

```cpp
#include <iostream>
#include <string>
using namespace std;
struct Distance //English distance
{
    int feet;
    float inches;

    void Display()
    {
        cout << feet << "' " << inches << "\"" << endl;
    }
};
struct Room //rectangular area
{
    Distance length; //length of rectangle
    Distance width; //width of rectangle
    float area;
    float Area(string units)
    {
        //convert length & width
        float l = length.feet + length.inches / 12;
        float w = width.feet + width.inches / 12;
        area = l * w;
        if (units == "english")
            return area;
        else if (units == "SI")
            return area * 0.092903;
    }
};
int main()
{
    Room dining; //define a room
    dining.length.feet = 13; //assign values to room
    dining.length.inches = 6.5;
    dining.width.feet = 10;
    dining.width.inches = 0.0;

    dining.length.Display();
    dining.width.Display();

    cout << "Dining room area is " << dining.Area("english") << " square feet\n";

    return 0;
}
```

Objects and Classes

- We already know that structures provide a way to group data elements together.
- We have also examined C++ functions, and how functions could interact with structures.
- Now, it is time for us to learn how to define objects that group both data (characteristics) and functions (behavior) together.
- These new entities are called **classes** .

Objects and Classes

- **`class smallObj`** defined in this program contains one **`private`** data item and two member functions **`set()`** and **`get()`**.

- These two member functions provide the *only* access to the variable **`data`** from outside of this **`class`**.

- Placing data and functions into a single entity (i.e. **`class`**) forms the essence of object-oriented programming.

```
#include <iostream>
using namespace std;
class smallObj //define a class
{
private:
    int data; //class data

public:
    void set(int d) //member function to set data (setter)
    {
        data = d;
    }
    int get() //member function to get data (getter)
    {
        return data;
    }
};
int main()
{
    smallObj s1, s2; //define two objects of class smallobj
    s1.set(1066); //call member function to set data
    s2.set(1776);
    cout << s1.get() << endl; //call member function to display data
    cout << s2.get() << endl;

    return 0;
}
```

Class Definitions

- An object is an **instance** of a **`class`**, in the same way that a BMW is an instance of a vehicle.
- In our previous example, s1 and s2 are **instances** of **`class smallObj`**.
- The **`class`** definition starts with the keyword **`class`** followed by the **`class`** name.
- Just like structures, the body of a **`class`** is enclosed in curly brackets **`{}`**.
- The body of the **`class`** is terminated by a semicolon **`;`**
- Recall that data constructs (e.g. classes and structures) are terminated by a semicolon whereas control structures (loops and functions) are not.

Private and Public

- The body of the **`class`** contains keywords **`private`** and **`public`**.
- A key feature of object-oriented programming is **data hiding** – this means that the data is concealed within a class so that accidental access by functions outside of the class is not possible.
- The main mechanism for hiding data is to include it in a class as a **`private`** data member.
- **`private`** data (and member functions) can only be accessed from within the class.
- **`public`** data (and member functions) can be accessed from outside.

Private and Public

- We should not confuse the principle of **data hiding** with security protocols used to protect computer databases.

- Data hiding means hiding data from parts of the program that do not need to access it.

- Specifically, one's class data is hidden from other classes – it is designed to protect well-intentioned programmers from honest mistakes.

- If a programmer really wants to find a way to access the data, it is still possible, **but difficult to do by accident**.

Private and Public

- Commonly, the data within a **`class`** is **`private`** and the member functions are **`public`**.

- This is a practice resulting from how classes are used – the data is hidden so it will be safe from accidental manipulation, while the member functions that manipulate the data are public.

- However, there is no rule that data must be **`private`** and functions **`public`**.

- In some circumstances, we may have to define **`private`** functions and **`public`** data (e.g. when you want to access the function only from within another member function of the class).

Member Access using the Member Access Operator

- To use a member function, the dot operator (i.e. the **member access operator**) is used (just as in structures):

```
s1.setData(123);
```

- We can think of calls to member functions as messages.

- It is as if we are sending a message to object **`s1`** to set or show its data when calling its member functions.

Class Constructors

- Member functions can be used to give values to data items in objects (i.e. initialize data members).
- However, it would be great if objects could initialize themselves when they are first created, without having to call a dedicated member function to do so.
- Automatic initialization is carried out by a special member function named a **Constructor**.
- A **Constructor is a member function** that is executed automatically whenever an object of a class is instantiated.
- If you do not provide a **`Constructor`** definition for a **`class`**, **a default `constructor` will automatically be defined for the class with no parameters**.

Simple Example: Counter

- Let us create a **`class`** of objects useful for counting something (e.g. the number of customers entering a store, the number of times a key is pressed, or any other event).

- Each time this event occurs, the **`counter`** is incremented by 1.

- Assume that this **`counter`** is important in the program and that it must be accessed by many different functions.

- We will use the object-oriented programming philosophy to define a **`class`** for **`counter`**.

Simple Example: Counter

- The **`counter class`** has a **`private`** data member **`count`**.
- When an object of this **`class`** is instantiated, we want to print a message onto the screen.
- The Constructor **`Counter()`** does this for us automatically every time an object of the class is instantiated.
- Our constructor does not accept any input parameters (**<u>no-parameter constructor</u>**).
- If we do not define a constructor, the compiler will define a default constructor for the class that does not do anything specific!

```
#include <iostream>
using namespace std;
class Counter
{
private:
    unsigned int count; //count
public:
    Counter() //no-parameter constructor
    {
        cout << "New object instantiated!" << endl;
    }

    void inc_count() //increment count
    {
        count++;
    }
    int get_count() //return count
    {
        return count;
    }
};
int main()
{
    Counter c1, c2; //define and initialize
    cout << "\nc1=" << c1.get_count(); //display
    cout << "\nc2=" << c2.get_count();
    c1.inc_count(); //increment c1’s count
    c2.inc_count(); //increment c2
    c2.inc_count(); //increment c2
    cout << "\nc1=" << c1.get_count(); //display again
    cout << "\nc2=" << c2.get_count();
    cout << endl;
    return 0;
}
```

Initializer List

- The preferred way of initializing data members inside a constructor is through an **initializer list**.
- The initialization occurs following the constructor declaration, but before the main body of the constructor.
- The value is placed inside brackets following the name of the data member after a colon.
- The initialization list is executed before the main body of the constructor is executed.
- As a result, it is the only way to initialize **`const`** data members.

```
#include <iostream>
using namespace std;
class Counter
{
private:
    unsigned int count; //count
public:
    Counter() : count(0) //default constructor
    {
        cout << "New Counter object is instantiated!" << endl;
    }

    void inc_count() //increment count
    {
        count++;
    }
    int get_count() //return count
    {
        return count;
    }
};
int main()
{
    Counter c1, c2; //define and initialize
    cout << "\nc1=" << c1.get_count(); //display
    cout << "\nc2=" << c2.get_count();
    c1.inc_count(); //increment c1
    c2.inc_count(); //increment c2
    c2.inc_count(); //increment c2
    cout << "\nc1=" << c1.get_count(); //display again
    cout << "\nc2=" << c2.get_count();
    cout << endl;
    return 0;
}
```

Initializer List

- If multiple data members must be initialized, they can be separated by commas.

```
MM() : m1(7.5), m2(33), m3(4) { /* empty constructor */ }
```

- Here, **`m1`**, **`m2`**, and **`m3`** are data members of **`class MM`**.
- This is called the ***member-initialization list*** or the ***initializer list****.*
- Array members can also be initialized using an initializer list.

```
someClass() : arr1{7.5, -1.0}, m2(33 { /* empty constructor */ }
```

Class Destructor

- We know that a special member function (i.e. the constructor) is called automatically when an object is first created.
- Another function that is called automatically, when an object is destroyed, is called a **Destructor** .
- A destructor has the same name as the constructor in your code (which is the same as the class name), but is preceded by a tilde **~**.
- We can define what we would like the destructor to do specifically, but if we don't, a destructor for the class will be created automatically.

Thank you.

DOUGLAS COLLEGE

DOUGLAS COLLEGE