# C++ Concepts

**Humayun Kabir**
Professor, CS, Vancouver Island University, BC, Canada

# Outline

- Constructor Function
- Destructor Function
- Const Function and Const Object
- Class and Member Function Prototyping
- Static Member Variable and Static Member Function
- Friend Function
- Friend Class
- Modular Programming

# Constructor

- **Constructor** function is a special ***member function*** of a class that is being automatically called to initialize the member variables of the class when an object is being created from the class.
- **Constructors** must have the **same name** as the class.
- Constructors have **no return type** (not even void).

```
class Account {
      private:
                  string acctNumber;
                  string acctOwner;
                  double acctBalance;
      public:
                  Account(string number, string owner, double balance):
                              acctNumber(number),
                              acctOwner(owner),
                              acctBalance(balance) {
                                          std::cout<<"Account::constructor()....."<<std::endl;
                              }
};
int main() {
                  Account bobAccount("10-001", "Bob", 119.0);                    //Calling constructor
                  Account* aliceAccount = new Account(“10-002", “Alice", 210.0);  //Calling constructor
                  return 0;
}
```

# Constructor

- **Constructors** function **cannot be static** or **const**
- Member variables are initialized using **member initialization list syntax**.
- The **body** of a constructor function is often **left empty**

```
class Account {
      private:
            string acctNumber;
            string acctOwner;
            double acctBalance;
      public:
            Account(string number, string owner, double balance) :
                  acctNumber(number),
                  acctOwner(owner),
                  acctBalance(balance) {
                        std::cout<<"Account::constructor()....."<<std::endl;
                  }
};
int main() {
            Account bobAccount("10-001", "Bob", 119.0);                    //Calling constructor
            Account* aliceAccount = new Account(“10-002", “Alice", 210.0);  //Calling constructor
            return 0;
}
```

# Constructor

- **Member initialization list syntax** is defined after the constructor parameters.
- Starts with a **colon**.
- Lists **each member variable** to **initialize** along with the **initialization value** using **direct initialization** for that variable, separated by a **comma**.
- Lists the **member variables** in the member initializer list **in the order** in which they have been **defined inside** the **class**.

```
class Account {
    private:
        string acctNumber;
        string acctOwner;
        double acctBalance;
    public:
        Account(string number, string owner, double balance) :
                acctNumber(number),
                acctOwner(owner),
                acctBalance(balance) {
                        std::cout<<"Account::constructor()....."<<std::endl;
                }
};
```

# Default Constructor

- **Default Constructor** is a **constructor** with **no parameter** or **argument**
- Member **variables** are **initialized** with some **default values**.

```
class Point {
      private:
                int _x;
                int _y;
      public:
                Point() :
                          _x(0),
                          _y(0)  {
                                    std::cout<<“Point::default-constructor()....."<<std::endl;
                          }
};
int main() {
                Point p1;                          //Calling default constructor
                Point* p = new Point();            //Calling default constructor
                return 0;
}
```

# Implicit Default Constructor

- **Implicit Default Constructor** is added by the compiler if no **constructor** is defined for the class
- It has empty member initialization list and empty body.

```
class Point {
      private:
            int _x;
            int _y;
      public:
            //Point() : {}                  //Implicit default constructor will be added by compiler
};
int main() {
            Point p1;                       //Calling implicit default constructor
            Point* p = new Point();         //Calling implicit default constructor
            return 0;
}
```

# Constructor Overloading

- Implicit default constructor **will not** be **added** by the compiler if a **constructor** is **defined** for the class.
- Default constructor is necessary if a class object is used to define a **member variable** of **another class**.
- Default constructor is also necessary to define **an array** of the **class type**.
- **Constructor** can be overloaded to define an **explicit default constructor** with one more **constructors** of the class.

```
class Point {
    private:
        int _x;
        int _y;
    public:
        Point() :
            _x(0),
            _y(0) { std::cout<<"Point::default-constructor()....."<<std::endl; }
        Point(int x, int y) :
            _x(x),
            _y(y)  { std::cout<<"Point::constructor()....."<<std::endl;}
};
```

# Constructor with Default Arguments

- **Constructor** with **all parameters** with **default arguments** can be called without any argument, i.e. is also a substitute of an **explicit default constructor**.

```
class Point {
      private:
                int _x;
                int _y;
      public:

                Point(int x=0, int y=0) :
                                _x(x),
                                _y(y)  {
                                                std::cout<<“Point::constructor()....."<<std::endl;
                                }
};
```

# Destructor

- **Destructor** function is a special ***member function*** of a class that is being automatically called to destroy an object of the class type when the object goes out of the scope or when delete is called on a class pointer.
- Destructors must have the **same name** as the **class** and precedes with **tilde**.
- Destructors have **no return type** (not even void) and **no parameter**.

```
class Account {
      private:
                  string acctNumber;
                  string acctOwner;
                  double acctBalance;
      public:
                  Account(string number, string owner, double balance):
                              acctNumber(number),
                              acctOwner(owner),
                              acctBalance(balance) {std::cout<<"Account::constructor()....."<<std::endl; }
                  ~Account() { std::cout<<“Account::destructor()……”<<std::endl; }
};
int main() {
                  Account bobAccount("10-001", "Bob", 119.0);                    //Calling constructor
                  Account* aliceAccount = new Account(“10-002", “Alice", 210.0);  //Calling constructor
                  //destructors for bobAccount and aliceAccount being called
                  delete aliceAccount;
                  return 0;
}
```

# Destructor

- **Destructor** function **cannot** be **static** and **cannot** be **const**.
- Destructor function **cannot** be **overloaded**.
- If no destructor is defined for a class an **implicit destructor** will be added by the compiler and the **body** of the implicit destructor is **empty**.

```
class Account {
      private:
                  string acctNumber;
                  string acctOwner;
                  double acctBalance;
      public:
                  Account(string number, string owner, double balance):
                              acctNumber(number),
                              acctOwner(owner),
                              acctBalance(balance) { std::cout<<"Account::constructor()....."<<std::endl; }
                  //~Account() {}              //Implicit destructor will be added by the compiler
};
int main() {
                  Account bobAccount("10-001", "Bob", 119.0);                          //Calling constructor
                  Account* aliceAccount = new Account(“10-002", “Alice", 210.0);   //Calling constructor
                  //destructors for bobAccount and aliceAccount being called
                  delete aliceAccount;
                  return 0;
}
```

# Explicit Destructor

- Most often the body of the destructor function can be left empty.
- If class object has **dynamic memory allocation** either by its **constructors** or by other **functions** those **memory** has to be **released** by the **body** of the **destructor** function**.**
- An **explicit destructor** with **non empty body** is necessary to avoid **memory leak** by the class objects.

```
class  GradePointAvergae{
      private:
                  int _capacity;
                  int _count;
                  double* _grades;
      public:
                   GradePointAvergae(int capacity):
                              _capacity(capacity),
                              _count(0),
                              _grades(newdouble[_capacity]) { }

                  ~GradePointAvergae () { delete [] _grades; }
};
int main() {
                  GradePointAverage bobGPA(40);                    //Calling constructor
                  //destructors for bobGPA being called and memory from bobGPA._grades has been released
                  return 0;
}
```

# Const Function and Const Object

- The **state** (**member variables**) of a **const object** of a class is **initialized** when it is being created and are **not allowed** to be **modified**.
- Any function of the class that is **not declared** as a **const function** in the class is **not allowed** to be **invoked** on a **const object** as it has potential to modify the state of the object**.**
- A function is **defined** as a **const function** by using **const keyword** after the parameter list.
- The **body** of the **const function** is **allowed to use** the **member variables** of the object **but not to modify** them.
- Only the **const functions** are **allowed** to be **invoked** on the **const object** of a class.
- A **const function** can also be **invoked** on a **non-const object**.

# Const Function and Const Object

```
class Account {
      private:
                  string acctNumber;
                  string acctOwner;
                  double acctBalance;
      public:
                  Account(string number, string owner, double balance) :
                              acctNumber(number),
                              acctOwner(owner),
                              acctBalance(balance) {
                                          cout<<"Account::constructor()....."<<endl;
                              }
                  string getAcctNumber() const { return acctNumber; }

                  string getAcctOwner() const { return acctOwner; }

                  double getAcctBalance() const { return acctBalance; }
};
int main() {
                  const Account constAccount("11-111", “Const", 119.0);          //State is initialized
                  std::cout<<constAccount.getAcctNumber()<<“, “<<constAccount.getAcctOwner()<<“, “<<
                              constAccount.getAcctBalance()<<std::endl;
                  return 0;
}
```

# Class and Member Functions Prototype

- Most of the examples shown so far have the member functions declared and defined within the class body. This is called **inline definition** of member functions.
- You can also declare and define your class member functions **non-inline**.
- You can declare the **prototype** of your **member functions** inside your **class body**.
- Your **class definition** will be **incomplete** unless you define or implement the prototyped member functions**.**
- You can **define** or implement **prototyped** member functions **outside** your **class body** within your **class scope**.
- You need to add **class name** and **scope operator** (**: :**) before the **function name** in order to implement your member function within your **class scope**.

# Prototyped Member Functions

```
class Account {
      private:
            string acctNumber;
            string acctOwner;
            double acctBalance;
      public:
            Account(string number, string owner, double balance);
            ~Account();
            string getAcctNumber() const;
            string getAcctOwner() const;
            double getAcctBalance() const;
            void deposit(double amount);
            void withdraw(double amount);
};
```

# Member Functions Outside Class Body

```
Account::Account(string number, string owner, double balance):
        acctNumber(number),
        acctOwner(owner),
        acctBalance(balance) {
                cout<<"Account::constructor()....."<<endl;
        }
Account::~Account(){
        cout<<"Account::destructor()....."<<endl;
        }
string Account::getAcctNumber() const {
        return acctNumber;
}
string Account::getAcctOwner() const {
        return acctOwner;
}
double Account::getAcctBalance() const {
        return acctBalance;
}
void Account::deposit(double amount) {
        acctBalance += amount;
}
void Account::withdraw(double amount) {
        acctBalance -= amount;
}
```

# Static Member Variable

- Each **object** of a **class** has its **own copies** of the **member variables** and for this reason, modifying the member variables of one object does not modify the member variables of the other objects of the same class.
- In order to **share** a **member variable** by **all the objects** of **a class**, you can declare a member variable **static** by using **static** keyword before its type.
- A **static member variable exists** before any object is created from a class.
- A **static member variable** can be **accessed** either on the **class** or on **any object** of the class.
- A **static member variable** is usually **initialized** by redefining it as a **global variable** on the class scope.
- Any **modification** of a **static member variable** is **visible** to **all the objects** of the class.

# Static Member Function

- You can also declare a **member function static** by using **static** keyword before its return type to manipulate **static member variables** of a class.
- A **static member function** also **exists** before any object is created from a class.
- A **static member function** can also be **invoked** either on the **class** or on **any object** of the class.
- A **static member function body** can access any **static member variable** but **no** non-static member variable of the class.
- A **static member function body** can invoke any other **static member function** but **no** non-static member function of the class.
- As ***this*** pointer is a self-reference to an object, ***this*** pointer is **not available** in the **body** of a **static member function**.
- A **static member function** of a class **cannot** be a **const function**.
- A **no-static member function** can **access** any **static member variable** and can **invoke** any **static member function**.

# Static Member Variable and Static Member Function

```
class Student {
        private:
                static int _id_tracker;                         //static member variable
                string _id;
                string _name;

                static string id(string prefix) {               //static member function
                      _id_tracker++;
                      return prefix+to_string(_id_tracker);
                }
                string id() const { return _id; }               //Overloaded non-static id() function

                string name() const { return _name; }
      public:
                Student(const string prefix, const string name): _id(id(prefix)), _name(name) {}
};
int Student::_id_tracker = 0;                   //Initializing static member variable
int main() {
                string prefix = "CSCI00";
                Student student1(prefix, "Humayun");
                Student student2(prefix, "Kabir");
                std::cout<<“id: “<<student1.id()<<“, name: “<<student1.name()<<std::endl;
                std::cout<<“id: “<<student2.id()<<“, name: “<<student2.name()<<std::endl;
                return 0;
}
```

# Friend Function

- You can declare a **friend function** of a class by using **friend** keyword before its return type in order to give **access** to the **private members** of the class.
- A **friend function** is **not** a **member function** of the class and does **not need** to **define** its body on the **class scope**.
- A **friend function** can be an **independent function** or a member function of another class.
- At least one **parameter** of the **friend function** must be **class type**.
- A **friend function** is **neither** private **nor** public irrespective of its position (private block or public block) inside class body

# Friend Function

```
class Point {
    private:
        int _x;
        int _y;
    public:
        Point(int x=0, int y=0) :
                _x(x),
                _y(y)  {
                        std::cout<<“Point::constructor()....."<<std::endl;
                }

        friend void show(const Point& p);
};

void show(const Point& p) {
        std::cout<<“<“<<p._x<<“,”<<p._y<<“>”<<std::endl;
}

int main() {
        Point p1(3,4);
        show(p1);
        return 0;
}
```

# Friend Class

- You can **make** a **member function** of **another class** a **friend function** of your class the **same way** you make an independent function a friend function of your class.
- If you need to **give access** to the **private members** of **your class** to **all** the **member functions** of **another class**, it is better to make that class a **friend class** of your class.
- You can make **another class** your **friend class** using **friend** keyword.
- Your class will **not automatically** become a **friend class** of the class that you have made your friend class. That class has to make your class a friend class explicitly, if necessary.

# Friend Class

```
class Node {
    private:
        int data;
        Node* next;
    public:
        Node():data(0), next(NULL) {}
        Node(int data): data(data), next(NULL) {}
        int getData() {return data;}
        friend class LinkedList;
};
```

```
class LinkedList {
    private:
        Node* head;
    public:
        LinkedList():head(new Node) {}
        ~LinkedList();
        void append(int data);
        bool remove(int data);
        Node* search(int data);
        void show();
};
```

# Friend Class

```
LinkedList::~LinkedList() {
    Node* current = head->next;
    while(current!= NULL) {
        Node* temp = current;
        current = current->next;
        delete temp;
    }
    delete head;
    cout<<"LinkedList destructor ...."<<endl;
}
```

# Friend Class

```
void LinkedList::append(int data) {
    Node* newNode = new Node(data);
    Node* current = head->next;
    if(current == NULL) {
        head->next = newNode;
    }
    else {
        while(current->next != NULL) {
            current = current->next;
        }
        current->next = newNode;
    }
}
```

# Friend Class

```
Node* LinkedList::search(int data) {
    Node* current = head->next;
    while(current!= NULL) {
        if(current->data == data){
            return current;
        }
        current = current->next;
    }
    return NULL;
}
```

# Modular Programming

- Most of the examples shown so far have **codes** for **class declaration**, **member function definition**, and **using class objects** in the same file. This **non-modular** approach is okay with **smaller applications** with smaller number of smaller classes.
- **Real** life **applications** are **big**, they are composed of hundreds of large classes, and they have to be programmed in a **modular** approach.
- In modular approach, you can declare the **prototype** of **your class** in a **header** file.
- You can **define** or implement the **member functions** of your class in a separate **implementation file** by including the **header** file of your class.
- You can **use objects** from your **class** in a **separate file** again by including the **header** file of your class.

# Modular Programming: Header File

```
#ifndef __CIRCLE_HEADER__
#define __CIRCLE_HEADER__

class Circle {

private:
        double radious;

public:
        Circle ();
        Circle(double radious);
        ~Circle();
        double getCircum();
        double getArea();
};

#endif
```

# Modular Programming: Implementation File

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

using namespace std;

Circle::Circle(): radious(0.0) {}

Circle::Circle(double radious): radious(radious) { }

Circle::~Circle() {
        cout<< "Circle destructor is being called....."<<endl;
}

double Circle::getCircum() {
        return 2.0*3.14*radious;
}

double Circle::getArea() {
        return 3.14*radious*radious;
}
```

# Modular Programming: Application File

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

using namespace std;

int main() {
		Circle foo(10.0);
		cout<<"foo circumference: "<<foo.getCircum()<<endl;
		cout<<"foo area: "<<foo.getArea()<<endl;
		Circle bar = 20.0;
		cout<<"bar circumference: "<<bar.getCircum()<<endl;
		cout<<"bar area: "<<bar.getArea()<<endl;
		Circle baz {30.0}; //uniform initializer
		cout<<"baz circumference: "<<baz.getCircum()<<endl;
		cout<<"baz area: "<<baz.getArea()<<endl;
		Circle qux = {40.0}; //uniform initializer
		cout<<"qux circumference: "<<qux.getCircum()<<endl;
		cout<<"qux area: "<<qux.getArea()<<endl;
		return 0;
}
```