Lecture Two

Encapsulation / Data Hiding

Implemented through Class

© **Dr. Mohammad Mahfuzul Islam, PEng**
Professor, Dept. of CSE, BUET

Constructor

- Object Creation may require some sort of ***initialization***. ***Constructor Method*** performs the tasks of initialization.
- A class's constructor is **called automatically** when an object of that class is created.
- A constructor function has the **same name as the class** of which it is a part and has **no return type**.
- Constructor may have or may not have **arguments**.

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

class MyClass {
      int a = 0;
public:
      MyClass(){cout << "No Argument Constructor" << endl;}
      MyClass(int newA); // constructor
      void show();
};

MyClass::MyClass(int newA){
      cout << "Constructor with Argument" << endl;
      a = newA;
}
```

C++ Code

```
void MyClass::show(){
      cout << a << '\n';
}

int main(){
      MyClass ob1, ob2(10);

      ob1.show();
      ob2.show();
      return 0;
}
```

Output:

```
No Argument Constructor
Constructor with Argument
0
10
```

Constructor

```
class MyClass {
    private int  a = 0;
    MyClass(){System.out.println("No Argument Constructor");}
    MyClass(int newA){
        System.out.println("Constructor with Argument");
        a = newA;
    }
    public void show() {
        System.out.println(a);
    }
}

public class Main {
    public static void main(String[] args){
        MyClass ob1 = new MyClass();
        MyClass ob2 = new MyClass(10);

        ob1.show();
        ob2.show();
    }
}
```

Java Code

Output:

```
No Argument Constructor
Constructor with Argument
0
10
```

Constructor – Object Cloning

C++ Code

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

class Box{
   double length;
   double width;
   double height;
public:
   Box(double l, double w, double h){
        length = l;
        width = w;
        height = h;
   }
   Box(const Box& ob){
        length = ob.length;
        width = ob.width;
        height = ob.height;
   }
   double volume(){
        return length * width * height;
   }
};
```

```
int main() {
   Box myBox(5, 3, 2);
   Box myClone(myBox);

   cout << "Box Volume: " << myBox.volume() << endl;
   cout << "Clone Volume: " << myClone.volume() << endl;
}
```

Output:

```
Box Volume: 30
Clone Volume: 30
```

Constructor – Object Cloning

Java Code

```
class Box{
    private double length;
    private double width;
    private double height;

    public Box(double l, double w, double h){
        length = l;
        width = w;
        height = h;
    }

    public Box(Box ob){
        length = ob.length;
        width = ob.width;
        height = ob.height;
    }

    public double volume(){
        return length * width * height;
    }
};
```

```
public class Main {
    public static void main(String[] args) {
        Box myBox = new Box(5, 3, 2);
        Box myClone = new Box(myBox);

        System.out.println("Box Volume: " + myBox.volume());
        System.out.println("Clone Volume: " + myClone.volume());
    }
}
```

Output:

```
Box Volume: 30
Clone Volume: 30
```

Destructor

- While working with object, some actions may be performed when an object is destroyed, e.g., freeing the memory allocated by the object. This destructor method is called when an **object is destroyed**.
- Local objects are destroyed when they go out of scope. Global objects are destroyed when the program ends.

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

class myclass {
    int a;
public:
    myclass();
    ~myclass(){cout << "Destructing……\n";};
    void show(){cout << a << endl;};
};
```

```
myclass::myclass(){
   cout << "In constructor\n";
   a = 10;
}

int main(){
   myclass ob;
   ob.show();
   return 0;
}
```

Output:

```
In constructor
10
Destructing……
```

- ✓ Java doesn't have any Destructor method.
- ✓ The garbage collector inherently handles memory allocation and release.

Object Pointer

- Object pointer stores **address** of the object.
- When a pointer to the object is used, the **arrow operator** (→) is employed rather than dot (.) operator.
- Like pointers to other types, an object pointer, when **incremented**, will point to the next object of its type.

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

class MyClass {
   int a;
public:
   MyClass(){ a = 10;}
   MyClass(int x){ a = x;};
   int getA() {return a;};
};
```

```
int main(){
   MyClass ob(120);
   MyClass *p;

   p = &ob;
   cout << "Object Value:" << ob.getA() << endl;
   cout << "Value using pointer:" << p->getA() << endl;

   return 0;
}
```

Output:

```
Object Value:120
Value using pointer:120
```

There is no **pointer operator** in Java.

The this Pointer / Reference

- A special pointer (C++) or reference (Java) called this represents the working object and is automatically passed to any member function when it is called.
- No programmer uses the this pointer / reference to access a class member because the shorthand form is much easier.

C++ Code

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

class Message {
  string msg;
public:
  Message(string msg) {
    this->msg = msg;
  }
  Message updateMessage(string msg);
  void displayMessage();
};

Message Message::updateMessage(string msg) {
  this->msg = msg;
  return *this;
}

void Message::displayMessage() {
  cout << "Message: " << this->msg << endl;
}

int main() {
  Message msg("Hello, World!");

  msg.displayMessage();
  msg = msg.updateMessage("New Message.");
  msg.displayMessage();
}
```

Output:

```
Message: Hello, World!
Message: New Message.
```

The this Pointer / Reference

The **this** keyword in Java is a reference variable that refers to the current object.

a = newA; => this.a = newA;

```
class Message {
    private String msg;

    public Message(String msg) {
        this.msg = msg;
    }

    public Message updateMessage(String msg) {
        this.msg = msg;
        return this;
    }

    public void displayMessage() {
        System.out.println("Message: " + this.msg);
    }
}
```

Java Code

```
public class Main {
    public static void main(String[] args) {
        Message msg = new Message("Hello, World!");

        msg.displayMessage();
        msg = msg.updateMessage("New Message.");
        msg.displayMessage();
    }
}
```

Output:

```
Message: Hello, World!
Message: New Message.
```

"**this**" means "**current object**".

Using new and delete in C++ Pointer

- C++ uses new operator for dynamically allocating memory (C uses malloc()).

  General form: p-var = new type;

  p-var = new type (initial value);

  p-var = new type [size];

- C++ uses delete operator for releasing dynamically allocating memory (C uses free()).

  General form: delete p-var;

  delete [] p-var;

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

class Samp{
    int a, b;
public:
    Samp(){ a = 0; b = 0;}
    Samp(int x, int y){ a = x; b = y;}
    int getProduct(){ return a * b;}
};
```

```
int main(){
    Samp *p = new Samp;
    Samp *q = new Samp(3, 4);
    Samp *r = new Samp[2];
    cout << "Product: " << p->getProduct() << endl;
    cout << "Product: " << q->getProduct() << endl;
    for(int i = 0; i < 2; i++){
        r[i] = Samp(i+1, i+2);
        cout << r[i].getProduct() << endl;
    }
    delete p;
    delete q;
    delete[] r;
    return 0;
}
```

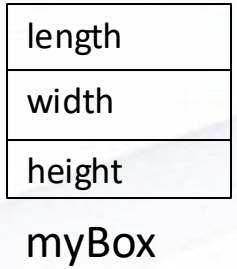
Output:

```
Product: 0
Product: 12
2
6
```

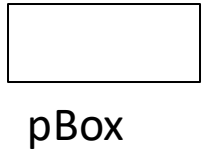
Object Declaration

C++

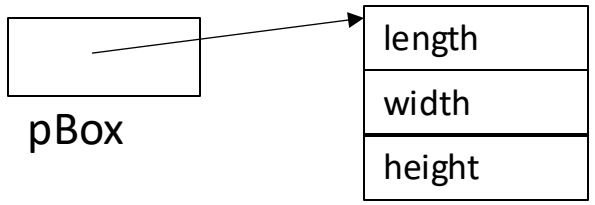
Box myBox;



Box *pBox;

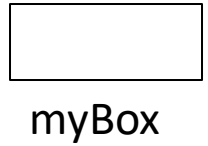


Box *pBox = new Box;

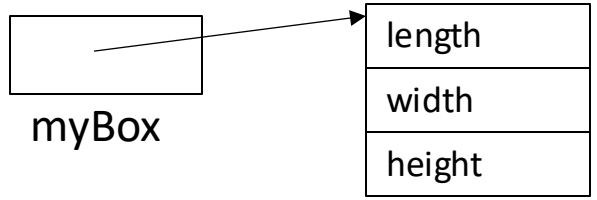


Java

Box myBox;



Box myBox = new Box();



Reference in C++

➢A reference is an implicit pointer that acts like another name of a variable. A reference can be used in Three ways:

- ✓A reference can be passed to a function
- ✓A reference can be returned by a function
- ✓An independent reference can be created.

Use of Pointer

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

void f(int *p) {
   *p = 100;
}

int main(){
   int a = 10;
   f(&a);
   cout << "a: " << a << endl;
   return 0;
}
```

C++ Code

Passing Reference to a function

Use of Reference

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

void f(int &p) {
   p = 100;
}

int main(){
   int a = 10;
   f(a);
   cout << "a: " << a << endl;
   return 0;
}
```

Output:

```
a: 100
```

Reference in C++

Passing Reference to a function

- When a reference parameter is used, the compiler automatically passes the address of the variable as the argument. There is no need to manually generate the address of the argument by preceding it with an & (in fact, it is not allowed).
- Within the function, the compiler automatically uses the variable pointed to by the reference parameter, no need to employ *.
- A reference parameter fully automates the call-by-reference parameter passing mechanism.

C++ Code

```
void f(int  &n){
    n = 100;
    n++;
}
```

- In the above example, instead of incrementing n, this statement increments the value of the variable being referenced (in this case, a).

Reference in C++

A Reference returned by a function

➢Very useful for overloading certain types of operator.

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

class Coord{
  int x, y;
public:
  Coord(int a, int b){x = a; y = b;}
  Coord& operator++(){
    x++;
    y++;
    return *this;
  }
  void show() {
    cout << "(" << x << "," << y << ")" << endl;
  }
};
```

```
int main(){
  Coord P(10, 20);
  P.show();
  ++P;
  P.show();
  return 0;
}
```

Output:

```
(10,20)
(11,21)
```

C++ Code

➢Allow a function to be used on the left side of an assignment statement.

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

int x;

int &f(){
   return x;
}

int main(){
   f() = 100;
   cout << x << endl;
   return 0;
}
```

Output:

```
100
```

BUT

```
int &f(){
    int x;
    return x;
}
```

Reference in C++

Independent Reference

- An independent reference is a reference variable that in all effects is simply another name for variable.
- Because reference cannot be assigned new values, an independent reference must be initialized when it is declared.

C++ Code

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

int main(){
  int x;
  int &ref = x;

  x = 10;
  ref = 100;
  cout << x << ", " << ref << endl;

  return 0;
}
```

Output:

```
100, 100
```

The independent reference ref serves as a different name for x.

Independent reference cannot be a constant like

const int &ref = 10;

In-line Function

In-line Function: Like macros in C, In-line functions are not actually called, rather are expanded in line, at the point of each call.

- **Advantage**:
  - In-line function has **no overhead** associated with the function call and return mechanism, so much **faster** than the normal function calls.
  - In parameterized macros, it is easy to forget extra parentheses are needed. In-line function is a **structure way to expand** short function in line and prevent the problems of parameterized macros.
- **Disadvantage**: If in-line functions are too large and called too often, **program grows larger**. Therefore, only short functions are declared in-line.

C++ Code

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

inline int even(int x){
    return (x % 2 == 0);
}

int main(){
    if (even(10))
        cout << "Even" << endl;
    return 0;
}
```

Output:

```
Even
```

Inline specifier is a request, not a command to the compiler:

Some compiler will not inline a function if it contains a static variable, loop, switch or goto. If any inline restriction is violated, the compiler is free to generate a normal function.

Automatic In-line Function

Automatic In-line Function: If a member's function definition in a class is short enough, then the function automatically becomes an in-line function. The **inline** keyword is no longer necessary.

- ✓ The same restriction that apply to a normal in-line functions apply to automatic in-line functions within a class declaration.

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

class Samp{
    int a, b;
public:
    Samp(int n, int m){a = n; b = m;}
    int divisible(){ return !(a%b);}
};
```

```
int main(){
    Samp s1(10, 2), s2(10,3);

    if (s1.divisible())
        cout << "10 is divisible by 2" << endl;
    if (s2.divisible())
        cout << "10 is divisible by 3" << endl;

    return 0;
}
```

Output:

```
10 is divisible by 2
```

In-line function in Java

- Java doesn't support inline command. But in Java, the compiler can perform in-lining when the small final method is called.
- Because final methods can't be overridden by subclasses, and the call to a final method is resolved at compile time.

```
class Figure{
      final private double pi = 3.14159;
      private double radius;

      //   public void setPI(double newPI){ pi = newPI;}
      public double getPI(){ return pi;}
      public void setRadius(double newRadius){ radius = newRadius;}
      public double getRadius(){ return radius;}

     final public double circleArea(){return pi*radius*radius;}
}
```

JAVA Code

```
class Cylindar extends Figure{
        private double height;

        Cylindar(double newRadius, double newHeight){
                setRadius(newRadius);
                //  setPI(3.14159);
                height = newHeight;
        }

        /*  public double circleArea(){
            return pi*radius*radius;
            } */

       public double cylindarVolume(){
            return circleArea()*height;
        }
}
```

Object Assignment

- One object can be **assigned** to another provided that both objects are of the **same type**.
- By default, when one object is assigned to another, a **bitwise copy** of all the data members is made.

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

class MyClass{
    int a, b;
public:
    void setValue(int n, int m){a = n; b = m;}
    void show(){
        cout << a << ", " << b << endl;
    }
};

class YourClass{
    int a, b;
public:
    void setValue(int n, int m){a = n; b = m;}
    void show(){
        cout << a << ", " << b << endl;
    }
};
```

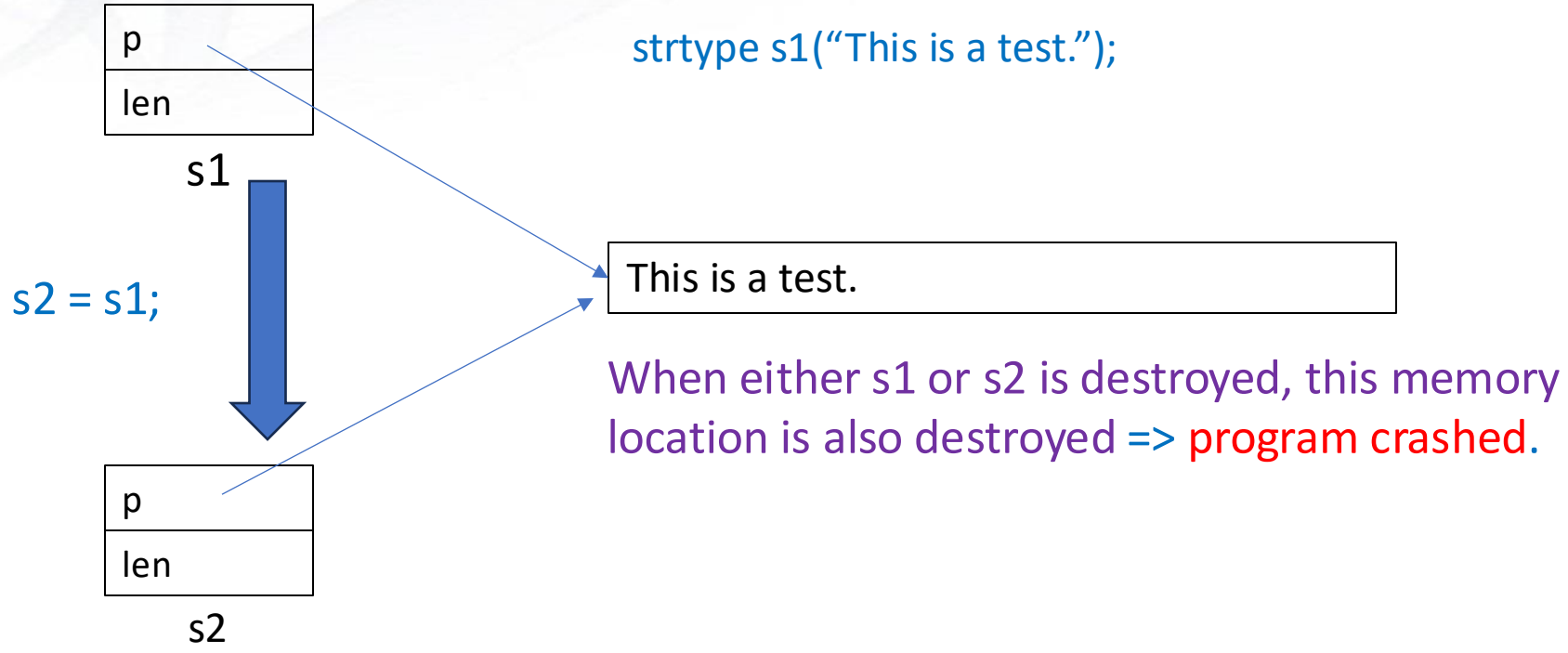
```
int main(){
    MyClass ob1, ob2;
    YourClass ob3;

    ob1.setValue(10, 20);
    ob2 = ob1; //Ok
    // ob3 = ob1; //Compilation Error
    ob1.show();
    ob2.show();

    return 0;
}
```

Output:

```
10, 20
10, 20
```

Problem with Object Assignment in C++

- When an object **pointing to dynamic memory allocation** is assigned to another object
  - both object **share** the **same memory**.
  - **Destroying** one object **release the common memory** and possibly cause **program crash**.



Problem with Object Assignment in C++

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

class ObAssign{
    char *str;
    int len;
public:
    ObAssign(char *s){
        len = strlen(s);
        str = (char *)malloc(len + 1);
        strcpy(str, s);
    }
    ~ObAssign(){
        cout << "Freeing memory" << endl;
        free(str);
    }
    void show(){
        cout << str << ":" << len << endl;
    }
};
```

```
int main(){
    ObAssign ob1("Hello World"), ob2("like C++");

    ob1.show();
    ob2.show();
    ob2 = ob1; // This will cause a problem
    ob2.show();

    return 0;
}
```

Program name:
ObAssignProblem.cpp

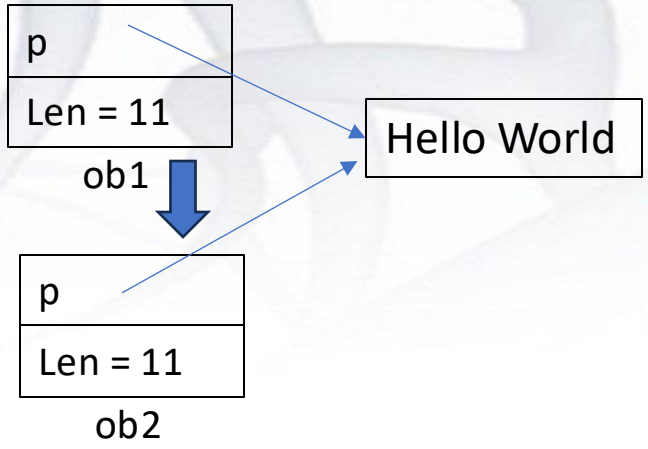
Output:

```
Hello World:11
Like C++:8
Hello World:11
Freeing memory
Freeing memory
ObAssignProblem(1217,0x109ed5600) malloc: *** error for
object 0x7f9842f059d0: pointer being freed was not
allocated
ObAssignProblem(1217,0x109ed5600) malloc: *** set a
breakpoint in malloc_error_break to debug
zsh: abort      ./"ObAssignProblem"
```

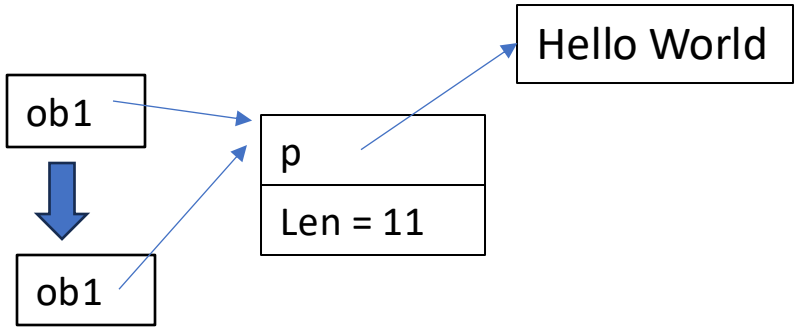
Solution to the Object Assignment Problem in C++

Solution 1: Use of Pointer

Without Pointer:



With Pointer:




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

class ObAssign{
    char *str;
    int len;
public:
    ObAssign(char *s){
        len = strlen(s);
        str = (char *)malloc(len + 1);
        strcpy(str, s);
    }
    ~ObAssign(){
        cout << "Freeing memory" << endl;
        free(str);
    }
    void show(){
        cout << str << ":" << len << endl;
    }
};
```

```
int main(){
    ObAssign *ob1 = new ObAssign("Hello World");
    ObAssign *ob2 = new ObAssign("Like C++");

    ob1->show();
    ob2->show();
    ob2 = ob1;
    ob2->show();

    return 0;
}
```

Output:

```
Hello World:11
Like C++:8
Hello World:11
```

No Execution of Destructor.

Acceptable??

Solution to the Object Assignment Problem in C++

Solution 2:
Copy Assignment Operator

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

class ObAssign {
    char *str;
    int len;
public:
    ObAssign(char *s) {
        len = strlen(s);
        str = (char *)malloc(len + 1);
        strcpy(str, s);
    }
    ObAssign& operator=(const ObAssign &obj) {
        if (this == &obj) {
            return *this; // self-assignment
        }                 // e.g., ob1 = ob1
        free(str);
        len = obj.len;
        str = (char *)malloc(len + 1);
        strcpy(str, obj.str);
        return *this;
    }
    ~ObAssign() {
        cout << "Freeing memory" << endl;
        free(str);
    }

    void show() {
        cout << str << ":" << len << endl;
    }
};

int main() {
    ObAssign ob1("Hello Word"), ob2("like C++");

    ob1.show();
    ob2.show();

    ob2 = ob1; // Assignment operator is invoked
    ob2.show();

    return 0;
}
```

Output:

```
Hello World:11
like C++:8
Hello World:11
Freeing memory
Freeing memory
```

Passing Object as an Argument in C++

- **Parameter passing,** by default, i**s called by value**. That is a bitwise copy is made.
- New object created in the function **does not call constructor,** but **destructor is called**.

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

class ObArg{
   int a;
public:
   ObArg(int n){
      a = n;
      cout << "Constructing..." << endl;
   }

   ~ObArg(){
      cout << "Destructing..." << endl;
   }

   void setA(int n){ a = n; }
   int getA(){ return a; }
};
```

```
void sqrOb(ObArg ob){
    ob.setA(ob.getA() * ob.getA());
    cout << "Inside sqrOb: " << ob.getA() << endl;
}

int main(){
    ObArg ob1(10);
    cout << "Before sqrOb: " << ob1.getA() << endl;
    sqrOb(ob1);
    cout << "After sqrOb: " << ob1.getA() << endl;
    return 0;
}
```

Output:

```
Constructing...
Before sqrOb: 10
Inside sqrOb: 100
Destructing...
After sqrOb: 10
Destructing...
```

Problem with Passing Object as an Argument in C++

- If the **object** used as the arguments **allocates dynamic memory** and **free** the memory then the destructor function is called and the **original object** is **damaged**.

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

class ObArgProb{
    int *p;
public:
    ObArgProb(int n){
        p = (int *)malloc(sizeof(int));
        *p = n;
        cout << "Constructing..." << endl;
    }
    ~ObArgProb(){
        cout << "Destructing..." << endl;
        free(p);
    }
    int getP(){ return *p; }
};

int negateP(ObArgProb ob){
    return -ob.getP();
}
```

```
int main(){
    ObArgProb ob1(10);

    cout << "Before: " << ob1.getP() << endl;
    cout << "Result: " << negateP(ob1) << endl;
    cout << "After: " << ob1.getP() << endl;
    return 0;
}
```

Output:

```
Constructing...
Before: 10
Result: -10
Destructing...
After: 10
Destructing...
ObArgProb(3214,0x11af7d600) malloc: *** error for object
0x7f933b004a50: pointer being freed was not allocated
ObArgProb(3214,0x11af7d600) malloc: *** set a breakpoint
in malloc_error_break to debug
zsh: abort      ./"ObArgProb"
```

Program name:
ObArgProb.cpp

Solution to Object Passing Problem in C++

Solution 1: Using Call by Reference

Solution 1(a): Using Pointer

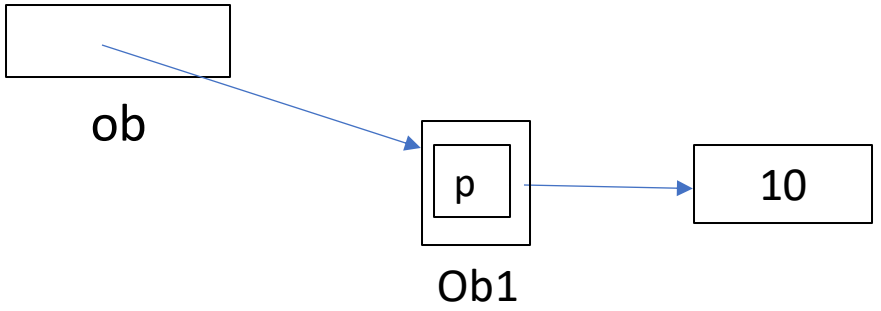
```
int negateP(ObArgProb *ob){
   return -ob->getP();
}

int main(){
   ObArgProb ob1(10);

   cout << "Before: " << ob1.getP() << endl;
   cout << "Result: " << negateP(&ob1) << endl;
   cout << "After: " << ob1.getP() << endl;

   return 0;
}
```

Output:

```
Constructing...
Before: 10
Result: -10
After: 10
Destructing...
```

Solution 1(b): Using Reference

```
int negateP(ObArgProb &ob){
     return -ob.getP();
}
```



Copy address of the Object, rather that copying the object itself.

Solution to Object Passing Problem in C++

Solution 2: Using Copy Constructor

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

class ObArgProb{
    int *p;
public:
    ObArgProb(int n){
        p = (int *)malloc(sizeof(int));
        *p = n;
        cout << "Constructing..." << endl;
    }

    // Copy constructor
    ObArgProb(const ObArgProb &ob){
        p = (int *)malloc(sizeof(int));
        *p = *(ob.p);
        cout << "Copying..." << endl;
    }
    ~ObArgProb(){
        cout << "Destructing..." << endl;
        free(p);
    }
    int getP(){ return *p; }
};

int negateP(ObArgProb ob){
    return -ob.getP();
}

int main(){
    ObArgProb ob1(10);

    cout << "Before: " << ob1.getP() << endl;
    cout << "Result: " << negateP(ob1) << endl;
    cout << "After: " << ob1.getP() << endl;

    return 0;
}
```

Output:

```
Constructing...
Before: 10
Result: Copying...
-10
Destructing...
After: 10
Destructing...
```

Object Returning Problem in C++

- When an object is **returned** by a function, a **temporary object** is **created** which holds the return value. This object is return by the function.
- After the **value** has been **returned**, this object is **destroyed**.
- The **destruction** of this **temporary object** may cause **unexpected side effects**.

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

class Samp {
    char *s;
public:
    Samp() { s = '\0'; }
    ~Samp() {free(s); cout << "Freeing S\n";}
    void show() {cout << s << endl;}
    void set (char *str){
        s = (char *) malloc(strlen(str)+1);
        strcpy(s, str);
    }
};
```

```
Samp input() {
    Samp s;
    s.set("Hello, world!");
    return s;
}
int main() {
    Samp ob;

    ob = input();
    ob.show();

    return 0;
}
```

Output:

```
Freeing S
Hello, world!
ObReturnProb(1337,0x106d8c600) malloc: *** error for
object 0x7f790e7059f0: pointer being freed was not
allocated
ObReturnProb(1337,0x106d8c600) malloc: *** set a
breakpoint in malloc_error_break to debug
zsh: abort      ./"ObReturnProb"
```

Solution to Object Returning Problem in C++

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

class Samp {
    char *s;
public:
    Samp() {s ='\0';}

    // Copy Assignment Operator
    Samp& operator=(const Samp &ob) {
        if (this == &ob) { return *this;}
        s = (char *)malloc(strlen(ob.s) + 1);
        strcpy(s, ob.s);
        return *this;
    }

    ~Samp() {free(s); cout << "Freeing S\n";}

    void show() {cout << s << endl;}
    void set(char *str) {
        s = (char *)malloc(strlen(str) + 1);
        strcpy(s, str);
    }
};

Samp input() {
    Samp s;
    s.set("Hello, world!");
    return s;
}

int main() {
    Samp ob;

    ob = input();
    ob.show();

    return 0;
}
```

Output:

```
Freeing S
Hello, world!
Freeing S
```

Use Java, Be Relaxed

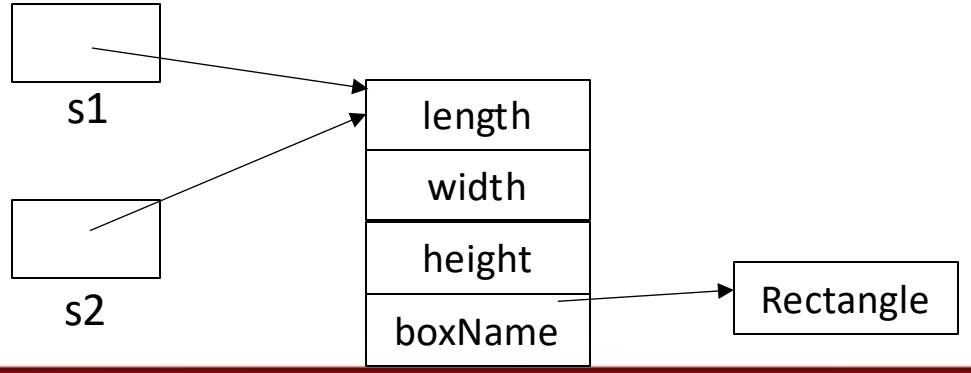
Java does not have -

- ✓ Object assignment problem
- ✓ Problem with passing object as argument
- ✓ Object returning problem

Automatic Argument Type Selection:

➢ When a primitive type argument is passed, then it is "**call by value**". Bitwise copy is made.

➢ When an object is passed to a method, it is "**Call by Reference**". Only creating a reference to the object.



Garbage Collection of Java: Java automatically manages memory allocation and deallocation for objects.

No Destructor in Java: Java does not have any destructors. Garbage collector inherently handle memory allocation and release.

- ➢ **All three C++ problems are arisen due to object destruction.**
- ➢ **As Java does not support object destruction, none of the problem exists in java.**

Friend Function in C++

- A friend function is **not a member** of a class but still has **access** to its **private elements**.
- **Three uses** of friend functions

  (1) to do **operator overloading**;

  (2) **creation** of certain types of **I/O functions**; and

  (3) one function to **have access** to the **private members** of **two or more different classes**.

- A friend function is a **regular non-member** function

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

class Truck;    // Forward Declaration

class Car{
    int passenger, speed;
public:
    Car(int p, int s){ passenger = p; speed = s;}
    friend int speedGreater(Car c, Truck t);
};
```

```
class Truck{
    int weight, speed;
public:
    Truck(int w, int s){ weight = w; speed = s;}
    friend int speedGreater(Car c, Truck t);
};

int speedGreater(Car c, Truck t){
    return c.speed - t.speed;
}

int main(){
    Car c(5, 70);
    Truck t(500, 60);

    cout << "Speed Gap: " << speedGreater(c,t) << endl;

    return 0;
}
```

Output:

```
Speed Gap: 10
```

Friend Function in C++

- A **member of one class** can be a **friend of another class**.

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

class Truck;      //Forward Declaration

class Car{
    int passenger, speed;
public:
    Car(int p, int s){ passenger = p; speed = s;}
    int speedGreater(Truck t);
};

class Truck{
    int weight, speed;
public:
    Truck(int w, int s){ weight = w; speed = s;}
    friend int Car::speedGreater(Truck t);
};
```

```
int Car::speedGreater(Truck t){
    return speed - t.speed;
}

int main(){
    Car c(5, 70);
    Truck t(500, 60);

    cout << "Speed Gap: " << c.speedGreater(t) << endl;

    return 0;
}
```

Java does not have a "friend" keyword or concept.

Output:

```
Speed Gap: 10
```

Use of static

- **static: Independent of object**. Static member can be accessed before creation of any object and without reference to any object.
  - ✓ Only **one copy of static variable** and all the objects of its class share it.
  - ✓ A static member cannot access non-static members of a class. **WHY?**
- Uses:

static variable: like global variable. When object is declared, no copy of static variable is made.

static method: Restrictions of static method-

- ✓ Can access static variables only and call only other static methods of their class.
- ✓ Can't refer to ***this*** or ***super*** anyway.
- ✓ **Static function** in **C** implies function can be used in multiple files.

static block: Executed exactly once, when first the class is loaded. Used for initialization.

- ✓ Supported only in Java. C & C++ does not support.

Use of static

Use of static in C

```
#include <stdio.h>
int a = 2;
static int b = 5;

static int mult(){
   a++; b++;
   return a * b;
}

void fun(){
   static int count = 0;
   count++;
   printf("Count: %d\n", count);
}

int main(){
   for(int i = 0; i < 5; i++){
      fun();
   }
   printf("Product: %d\n", mult());
   return 0;
}
```

Output:

```
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
Product: 18
```

Use of static (variable & method) in C++

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

class StaticDemo {
      static int a;
      // static int b = 5;
      static const int b = 5;
      int n = 4, m = 5;

public:
      //static void increment() {a++; b++;}
      static void increment() {a++;}
      // static int getProduct(){ return n * m;}
      int getProduct() { return n * m;}

      static void display() {
          cout << "A: " << a << " B: " << b << endl;
      }
};

int StaticDemo::a = 0;
```

```
int main() {
     StaticDemo sd;

     sd.display();
     sd.increment();
     StaticDemo::display();
     Cout << sd.getProduct() << endl;
     return 0;
}
```

Output:

```
A: 0 B: 5
A: 1 B: 5
20
```

Use of static

Use of static in Java

```
class StaticDemo {
    static int a = 3;
    static int b;
    static int c;

    static void show(int x) {
        a++;  b--;
        System.out.print("A: " + a + " ");
        System.out.print("B: " + b + " ");
        System.out.print("C: " + c + " ");
        System.out.println("X: " + x);
    }

    static {
        System.out.println("Static is Initialized.");
        b = a * 4;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        StaticDemo sd = new StaticDemo();
        StaticDemo.c = 5;

        for (int i = 0; i < 2; ++i) {
            sd.show(40 + i);
        }
        // show(10);
        StaticDemo.show(10);
        System.out.println("B:"+ StaticDemo.b);
    }
}
```

Output:

```
Static is Initialized.
A: 4 B: 11 C: 5 X: 40
A: 5 B: 10 C: 5 X: 41
A: 6 B: 9 C: 5 X: 10
B:9
```

Use of import static in Java

➢**Use of import static allows static members of a class or interface can be called without reference.**

```
import java.lang.Math.*;
import java.lang.System.*;

public class Main {
    public static void main(String[] args) {
        double s1 = 3.0, s2 = 4.0;
        int x1 = 5, y1 = 8, x2 = 6, y2 = 10;

        double h = Math.hypot(s1, s2);
        double dist = Math.sqrt(Math.pow(x1-x2, 2)+ Math.pow(y1-y2, 2));

        System.out.println("Hypotenuse: "+ h + "  Dist: " + dist);
    }
}
```

```
import static java.lang.Math.*;
import static java.lang.System.out;

public class Main {
    public static void main(String[] args) {
        double s1 = 3.0, s2 = 4.0;
        int x1 = 5, y1 = 8, x2 = 6, y2 = 10;

        double h = hypot(s1, s2);
        double dist = sqrt(pow(x1-x2, 2)+ pow(y1-y2, 2));

        out.println("Hypotenuse: "+ h + "  Dist: " + dist);
    }
}
```

OUTPUT:

Hypotenuse: 5.0 Dist: 2.23606797749979

Use of const and mutable in C++

- When a member function is declared as const, it cannot modify the object that invokes it.
- mutable overrides const-ness. A mutable member can be modified by a const member function.
- Non-Member function cannot be const.

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

class Demo {
    mutable int a;
    int b;

public:
    Demo(){a = 0; b = 0;}
    int getA() const {return a;}
    int getB() const {return b;}
    void setAB(int x, int y) const {
        a = x;
        // b = y;
    }
};
```

```
int main() {
    const Demo d;

    d.setAB(10, 20);
    cout << "A: " << d.getA() << endl;
    cout << "B: " << d.getB() << endl;

    return 0;
}
```

Output:

```
A: 10
B: 0
```

Use of final in Java

- Three uses of **final** keyword:

  final variable: constant, i.e., can't be modified.

  Can be initialized two ways:

  - ✓ When declared.

    final double PI =3.14159;

  - ✓ Assign a value within constructor.

  final method: can't be overridden.

  final class: can't be inherited.

C++ does not have "final" keyword.

```
public class Main {
    public static void main(String[] args) {
        Cylindar cylindar = new Cylindar(3.0, 4.0);
        System.out.print(cylindar.cylindarVolume());
    }
}
```

```
class Figure {
    final private double pi = 3.14159;
    private double radius;

    // public void setPI(double newPI){ pi = newPI;}
    public void setRadius(double newRadius) { radius = newRadius; }
    final public double circleArea() { return pi * radius * radius; }
}

final class Cylindar extends Figure {
    final private double height;

    Cylindar(double newRadius, double newHeight) {
        setRadius(newRadius);
        height = newHeight;
    }

    //  public double circleArea(){ return 3.14159*radius*radius; }
    public double cylindarVolume() { return circleArea() * height; }
}
```

Array of Objects in C++

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

class Sa{
    int a;
public:
    Sa(int n){ a = n;}
    int getA(){ return a;}
};

class Da{
    string name;
    int b;
public:
   Da(string nam, int n){ name = nam; b = n;}
   void show(){
        cout << "(" << name << ", " << b << ")" << endl;
   }
};
```

Output:

```
s1[0] = 10 s2[0] = 50
(A, 10)
s1[1] = 20 s2[1] = 60
(B, 20)
s1[2] = 30 s2[2] = 70
(C, 30)
s1[3] = 40 s2[3] = 80
(D, 40)
s3[0][0] = 11 s4[0][0] = 41
s3[0][1] = 12 s4[0][1] = 42
s3[1][0] = 21 s4[1][0] = 51
s3[1][1] = 22 s4[1][1] = 52
s3[2][0] = 31 s4[2][0] = 61
s3[2][1] = 32 s4[2][1] = 62
```

```
int main(){
    Sa s1[4] = {10, 20, 30, 40};
    Sa s2[4] = {Sa(50), Sa(60), Sa(70), Sa(80)};
    Sa s3[3][2] = { {11, 12}, {21, 22}, {31, 32} };
    Sa s4[3][2] = { {Sa(41), Sa(42)}, {Sa(51), Sa(52)}, {Sa(61), Sa(62)} };
    Da ob[4] = { Da("A", 10), Da("B", 20), Da("C", 30), Da("D", 40) };

    for (int i = 0; i < 4; i++){
        cout << "s1[" << i << "] = " << s1[i].getA() << " ";
        cout << "s2[" << i << "] = " << s2[i].getA() << endl;
        ob[i].show();
    }

    for (int i = 0; i < 3; i++){
        for (int j = 0; j < 2; j++){
            cout << "s3[" << i << "][" << j << "] = " << s3[i][j].getA() << " ";
            cout << "s4[" << i << "][" << j << "] = " << s4[i][j].getA() << endl;
        }
    }
}
```

Array of Objects in Java

```
class Samp {
    private int a;
    Samp(int n) { a = n;}
    public int getA(){ return a;}
}

class Std{
    String name;
    int id;
    Std(String n, int i){ name = n; id = i;}
    public void show(){
        System.out.println(name + ":" + id);
    }
}
```

Output:

```
40 41 42 43
J:101
L:102
H:103
```

```
public class Main {
    public static void main(String[] args) {
        int[] A = new int[4];
        int[] B = {11, 12, 13, 14};
        int[][] C = {{21, 22}, {23, 24}, {25, 26}};
        Samp[] S = new Samp[4];
        Std[] std = {new Std("J", 101), new Std("L", 102), new Std("H", 103)};

        for(int i =0; i < 4; ++i){
            A[i] = 30 + i;
            S[i] = new Samp(40+i);
        }

        for(int i =0; i< S.length; ++i){
            System.out.print(S[i].getA()+" ");
        }
        System.out.println();
        for(Std s: std){
            System.out.println(s.name + ":" + s.id);
        }
    }
}
```

Nested Class

- The scope of Nested Class is limited by the scope of the enclosing class.
- A Nested class is a static or non-static member of the enclosing class. The non-static nested class is known as *Inner Class*.
- A Inner Class class has access to the members of enclosing class (including private members); but the enclosing class does not have access to the members of the nested class.

C++ Code

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

class Outer {
    int a;
public:
    Outer(int x) { a = x; }
    void show() { cout << "Outer: " << a << endl; }

    void showInner(int x) {
        Inner inner(x);
        inner.show();
    }

    class Inner {
        int b;
    public:
        Inner(int y) { b = y; }
        void show() { cout << "Inner: " << b << endl; }
        void showOuter(Outer &ob) { ob.show(); }
    };
    class Inner2 {
    public:
        void show(string str) {
            cout << str << endl;
        }
    };
    Inner2 inner2;
};
```

```
int main() {
    Outer outer(10);
    outer.show();
    outer.showInner(20);

    Outer::Inner inner(30);
    inner.show();
    inner.showOuter(outer);
    outer.inner2.show("Hello from Inner2!");
    return 0;
}
```

Output:

```
Outer: 10
Inner: 20
Inner: 30
Outer: 10
Hello from Inner2!
```

Nested Class

```
class Outer {
        private int a;
        Outer(int x) { a = x; }
        public void show() { System.out.println("Outer: " + a);}
        public void showInner(int x) {
                Inner inner = new Inner(x);
                inner.show();
        }

        public class Inner {
                private int b;
                Inner(int y) { b = y; }
                public void show() { System.out.println("Inner: " + b); }
                public void showOuter(Outer ob) { ob.show(); }
        };

        public class Inner2 {
                public void show(String str) { System.out.println(str);}
        };
        Inner2 inner2 = new Inner2();
};
```

Java Code

```
public class Main {
        public static void main(String[] args) {
                Outer outer = new Outer(10);
                outer.show();
                outer.showInner(20);

                Outer.Inner inner = outer.new Inner(30);
                 inner.show();
                 inner.showOuter(outer);

                 outer.inner2.show("Hello from Inner2!");
        }
}
```

Output:

```
Outer: 10
Inner: 20
Inner: 30
Outer: 10
Hello from Inner2!
```

String Class in Java

- **String** type object is immutable. Once a **String** object is created, its contents will not be altered.

```
String myStr = new String("This is a test.");
String myStr = "This is a test.";
```

- Three methods of String
  - ✓ boolean equals(*secondStr*)
  - ✓ int length()
  - ✓ char charAt(*index*)

- **StringBuffer** and **StringBuilder** are peer classes of **String,** which allows string to be altered.

Command Line Arguments in Java

A Sample Program:

```
public class CommandLine {
    public static void main(String[] args) {
        for (int i = 0; i < args.length; i++) {
            System.out.println("args[" + i + "]: " + args[i]);
        }
    }
}
```

Command Line:

```
……> javac CommandLine
……> java CommandLine this is a test 100 -1
```

Output:

```
args[0]: this
args[1]: is
args[2]: a
args[3]: test
args[4]: 100
args[5]: -1
```

Local Variable Type Inference in Java

```
public class Main {
    public static void main(String[] args) {

        String [] str1 = {"one", "two", "three"};
        var str2 = "This is a test.";

        for (var i = 0; i < str1.length; i++) {
            System.out.println("str[" + i + "]: " + str1[i]);
        }

        for( var a: str1) System.out.println(a);
        System.out.println(str2);
    }
}
```

String

int

?

?

```
var fin = new FileInputStream("test.txt");
```

? Ambiguity? Why?

```
var x = o.getNext();
```