# 04: Class and Object Manipulations

Programming Technique II

(SCSJ1023)

*Adapted from Tony Gaddis and Barret Krupnow (2016), Starting out with C++: From Control Structures through Objects*

# Friends of Classes

A **friend class** is a **class** that can access the private and protected members of a **class** in which it is declared as **friend**. This is needed when we want to allow a particular **class** to access the private and protected members of a **class**.

# Friends of Classes

- **Friend**: a function or class that is not a member of a class, but has access to private members of the class
- A friend function can be a stand-alone function or a member function of another class
- It is declared a friend of a class with `friend` keyword in the function prototype

# Friends of Classes

- Stand-alone function:

```
friend void setAVal(intVal&, int);
// declares setAVal function to be
// a friend of this class
```

- Member function of another class:

```
friend void SomeClass::setNum(int num)
// setNum function from SomeClass
// class is a friend of this class
```

# Friends of Classes

- Class as a friend of a class:

```
class FriendClass
{
  ...
};
class NewClass
{
  public:
    friend class FriendClass;
   // declares entire class as a friend
   //  of this class FriendClass
  …
};
```

# Friends of Classes

```
// C++ program to demonstrate the working of friend class
#include <iostream>
using namespace std; // forward declaration
class ClassB;
class ClassA
{ private: int numA; // friend class declaration
friend class ClassB;
public: // constructor to initialize numA to 12
ClassA() : numA(12) {}
};
class ClassB
{ private: int numB;
public: // constructor to initialize numB to 1
ClassB() : numB(1)
{} // member function to add numA // from ClassA and numB from ClassB
int add()
{ ClassA objectA;
return objectA.numA + numB;
} };
int main()
{
ClassB objectB;
cout << "Sum: " << objectB.add();
return 0;
}
```

# Friends of Classes

```
#include <iostream>
using namespace std;
 class Box
{ double width;
  public:
  friend void printWidth( Box box );
  void setWidth( double wid ); };
// Member function definition
  void Box::setWidth( double wid )
{ width = wid; }
// Note: printWidth() is not a member function of any class.
  void printWidth( Box box )
{
/* Because printWidth() is a friend of Box, it can directly access any member
of this class */
cout << "Width of box : " << box.width <<endl; }
// Main function for the program
int main()
{ Box box; // set box width without member function
box.setWidth(10.0);
// Use friend function to print the wdith.
printWidth( box );
return 0; }
```

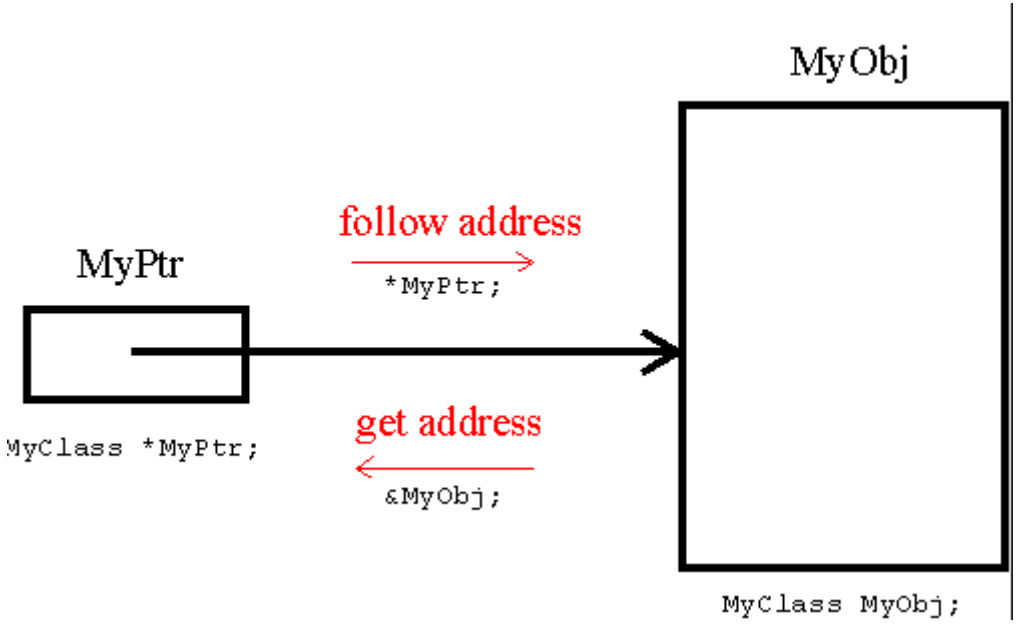
**Please watch this video:**

https://www.youtube.com/watch?v=WCFGNdXSzus

# FRIENDS OF CLASS

# Pointers to Objects

In C++, a **pointer** holds the address of an **object** stored in memory. The **pointer** then simply "points" to the **object**. The **type** of the **object** must correspond with the **type** of the **pointer.**



# Pointers to Objects

- You can declare a pointer to an object:

```
Rectangle *rPtr;
```

Declaring a pointer which can only point to an object of `Rectangle`

- Then, you can access public members via the pointer:

```
rPtr = &otherRectangle;
rPtrz->setLength(12.5);
(*rPtr).setLength(12.5);
cout << rPtr->getLenght() << endl;
```

`rPtr` is now pointing to the object `otherRectangle`

two types of syntax to access object's members via pointer.

# Pointers to Objects

Revisit the concept:

- A variable is meant to contain or hold a value.
- A **pointer** is meant to point to a variable (not to contain value).
  - Declaring a pointer does not create an object. Thus, no constructor is executed.

- Example:

```
Rectangle *p;
Rectangle r;
```

`p` is not an object but a pointer. No object is created here. Thus no constructor is executed.

`r` is an object of class `Rectangle`. The default constructor is executed here.

# Pointers to Objects

- The following declaration does not work: why?

```
Rectangle *p();
```

you may think of that a default constructor will be executed here. **Wrong!** A pointer does not have a constructor, only objects have.

In fact, the compiler assumes this code as a function prototype named **`p()`** which returns a pointer of type `Rectangle` i.e.

```
Rectangle* p();
```

https://www.youtube.com/watch?v=-GrsH3aKUfA

```
#include <iostream>
using namespace std;
class Box {
public:
// Constructor definition
Box(double l = 2.0, double b = 2.0, double h = 2.0)
{ cout <<"Constructor called." << endl;
length = l; breadth = b; height = h; }
double Volume()
{ return length * breadth * height; }
private: double length;
// Length of a box
double breadth;
// Breadth of a box
double height;
// Height of a box
};
int main(void)
{
// Declare box1
Box Box1(3.3, 1.2, 1.5);
// Declare box2
Box Box2(8.5, 6.0, 2.0);
// Declare pointer to a class.
Box *ptrBox;
// Save the address of first object
ptrBox = &Box1;
// Now try to access a member using member access operator
cout << "Volume of Box1: " << ptrBox->Volume() << endl;
// Save the address of second object
ptrBox = &Box2;
// Now try to access a member using member access operator
cout << "Volume of Box2: " << ptrBox->Volume() << endl;
return 0;
}
```

# Dynamically Allocating an Object

We can also use a **pointer** to **dynamically allocate an object**.

```
1   // Define a Rectangle pointer.
2   Rectangle *rectPtr;
3
4   // Dynamically allocate a Rectangle object.
5   rectPtr = new Rectangle;
6
7   // Store values in the object's width and length.
8   rectPtr->setWidth(10.0);
9   rectPtr->setLength(15.0);
10
11  // Delete the object from memory.
12  delete rectPtr;
13  rectPtr = 0;
```

# POINTERS TO OBJECTS

https://www.youtube.com/watch?v=-GrsH3aKUfA

# Arrays of Objects

# Arrays of Objects

```
class InventoryItem {
   private:
      char *description;  double cost; int units;
   public:
      InventoryItem();
      InventoryItem(const char desc[]);
      InventoryItem(const char desc[],double c, int u);
     ~InventoryItem();
      :
};  // end of class declaration
```

- Objects can be the elements of an array:

```
InventoryItem inventory[40];
```

- **Default constructor** for object is used when **array is defined**

# Arrays of Objects

- Must use initializer list to invoke constructor that takes arguments:

```
InventoryItem inventory[3] =
{"Hammer", "Wrench", "Pliers"};.
```

- If the constructor requires more than one argument, the initializer must take the form of a function call:

```
InventoryItem inventory[3] = { InventoryItem("Hammer", 6.95, 12),
                               InventoryItem("Wrench", 8.75, 20),
                               InventoryItem("Pliers", 3.75, 10) };
```

# Arrays of Objects

- It **isn't necessary** to call the **same constructor** for each object in an array:

```
InventoryItem inventory[3] = { "Hammer",
                               InventoryItem("Wrench", 8.75, 20),
                               "Pliers" };
```

- **Objects in an array** are referenced using **subscripts**

- **Member functions** are referenced using **dot notation**:

```
inventory[2].setUnits(30);
cout << inventory[2].getUnits();
```

# Example -Accessing Objects in an Array

**Program 1**

```
1  // This program demonstrates an array of class objects.
2  #include <iostream>
3  #include <iomanip>
4  #include "InventoryItem.h"
5  using namespace std;
6
7  int main()
8  {
9     const int NUM_ITEMS = 5;
10    InventoryItem inventory[NUM_ITEMS] = {
11                  InventoryItem("Hammer", 6.95, 12),
12                  InventoryItem("Wrench", 8.75, 20),
13                  InventoryItem("Pliers", 3.75, 10),
14                  InventoryItem("Ratchet", 7.95, 14),
15                  InventoryItem("Screwdriver", 2.50, 22) };
16
17    cout << setw(14) <<"Inventory Item"
18         << setw(8) << "Cost" << setw(8)
19         << setw(16) << "Units On Hand\n";
20    cout << "-----------------------------------\n";
```

# Example -Accessing Objects in an Array

**Program 1 (continued)**

```
21
22      for (int i = 0; i < NUM_ITEMS; i++)
23      {
24         cout << setw(14) << inventory[i].getDescription();
25         cout << setw(8) << inventory[i].getCost();
26         cout << setw(7) << inventory[i].getUnits() << endl;
27      }
28
29      return 0;
30   }
```

**Program Output**

```
Inventory Item    Cost  Units On Hand
---------------------------------------
        Hammer    6.95     12
        Wrench    8.75     20
        Pliers    3.75     10
       Ratchet    7.95     14
   Screwdriver     2.5     22
```

# Arrays of Objects (Example)

```
#include<iostream>
using namespace std;
class Arafat
{ private:
  string name; int age;
 public:
void getName() { cout<<"Your name is : " << name << "\n"; } void setName() { cout<<"Enter your name : ";
cin>>name; } int setAge() { cout<<"Enter your age : "; cin>>age; }
void getAge() { cout<<"Your age is : " << age << "\n"; } };
int main()
{ int size = 4;
Arafat array[size];
//Array holding 4 object references of type, Arafat.
//Setting the name and age of properties of each object of Arafat class, stored in an array

for(int i=0; i<size; i++)
//accessing array elements using length variable in for-loop
{ array[i].setName(); array[i].setAge(); }
//Getting the name and age of properties of each object of Arafat class, stored in an array

for(int i=0; i<size; i++)
//accessing array elements using length variable in for-loop
{ array[i].getName(); array[i].getAge(); } }
```

# ARRAYS OF OBJECTS

https://www.youtube.com/watch?v=eSpp3wXDMVE

# Objects and Functions

# Objects as  Function Parameters

# Passing Objects to Functions

- Can pass an object to a function in 3 ways:
  - Pass-by-value
  - Pass-by-reference
  - Pass-by-reference via pointer

To pass an object as an argument we write the object name as the argument while calling the function the same way we do it for other variables

```
function_name(object_name);
```

# Example 1: Pass-By-Value

```
#include <iostream>
using namespace std;
class Circle
{     private:    double radius;
      public:
            Circle(double r){radius=r;}
            double getRadius(){return radius;}
            double getArea(){return radius*radius*3.14;}
};
void printCircle(Circle a)
{     cout<<a.getRadius()<<" "<<a.getArea();}

int main()
{   Circle ab(5.5);
    printCircle(ab);
    return 0;
}
```

# Example 2: Pass-By-Value

```
class Count
{ public: int num;
          Count(int c){num = c;}
          Count(){num=0;}
};
void increment(Count c)
{    c.num++;   }

int main()
{ Count myCount;
  for(int i=0;i<10;i++)
     increment(myCount);
  cout<<myCount.num;
  return 0;
}
```

# Example 1: Pass-By-Reference

```
#include <iostream>
using namespace std;
class Circle
{     private:    double radius;
      public:
            Circle(double r){radius=r;}
            double getRadius(){return radius;}
            double getArea()
            {return radius*radius*3.14;}
};
void printCircle(Circle &a)
{     cout<<a.getRadius()<<" "<<a.getArea();}
int main()
{   Circle ab(5.5);
    printCircle(ab);
    return 0;
}
```

# Example 2: Pass-By-Reference

```
class Count
{ public: int num;
          Count(int c){num = c;}
          Count(){num=0;}
};
void increment(Count &c)
{    c.num++;   }

int main()
{ Count myCount;
  for(int i=0;i<10;i++)
          increment(myCount);
  cout<<myCount.num;
  return 0;
}
```

# Pass-By-Reference via Pointer

```
#include <iostream>
using namespace std;
class Circle
{     private:    double radius;
      public:
            Circle(double r){radius=r;}
            double getRadius(){return radius;}
            double getArea(){return radius*radius*3.14;}
};
void printCircle(Circle *a)
{     cout<<a->getRadius()<<" "<<a->getArea();}
int main()
{   Circle ab(5.5);
    printCircle(&ab);
    return 0;
}
```

# Example 2: Pass-By-Reference via Pointer

```
class Count
{ public: int num;
          Count(int c){num = c;}
          Count(){num=0;}
};
void increment(Count *c)
{     c->num++;    }
int main()
{ Count myCount;
  for(int i=0;i<10;i++)
          increment(&myCount);
  cout<<myCount.num;
  return 0;
}
```

# Returning Objects from Functions

**Syntax:**

```
return object_name;
```

https://www.youtube.com/watch?v=FX-g9Zwinlw

# Example

```
#include <iostream>
using namespace std;
class ClassName {
private:  int x, y;
public:
    ClassName readData()
  {
          ClassName temp;
          cout << "please input x and y "<<endl;
          cin>>x;
          cin>>y;
          temp.x=x+2;
          temp.y=y*3;
          return temp;
   }
           void display(){ cout << " x: " << x << endl << " y "       << y <<endl;
    }
};
int main()
{
    ClassName o1, o2;
    o2 = o1.readData();
    o1.display();
    o2.display();
    return 0;
}
```

# Operator Overloading

https://www.youtube.com/watch?v=DVMZPOt816E

# Operator Overloading

- Operators such as =, +, and others can be **redefined** when used with objects of a class

- The name of the function for the overloaded operator is `operator` followed by the operator symbol, *e.g.*,
  - `operator+` to overload the + operator, and
  - `operator=` to overload the = operator

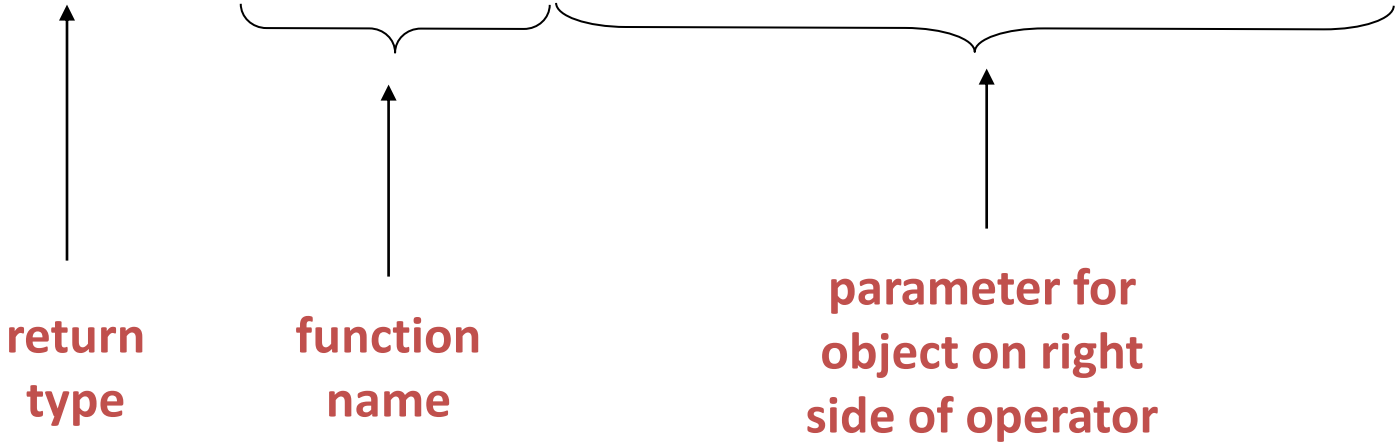
- Prototype for the overloaded operator goes in the declaration of the class that is overloading it

- Overloaded operator function definition goes with other member functions

# Operator Overloading

- Operators such as =, +, and others can be redefined when used with objects of a class. Prototype:

```
void operator=(const SomeClass &rval)
```




- Operator is called via **object on left side**

# Invoking an Overloaded Operator

- Operator can be invoked as a member function:

```
object1.operator=(object2);
```

- It can also be used in more conventional manner:

```
object1 = object2;
```

# Example: operator=

```
class PersonInfo
{
private:
  char *name;   int age;
public:
     PersonInfo(char *n, int a) // Constructor
     { name = new char[strlen(n) + 1];
       strcpy(name, n);     age = a; }

  // Copy Constructor
  PersonInfo(const PersonInfo &obj)
     { name = new char[strlen(obj.name) + 1];
       strcpy(name, obj.name);
       age = obj.age; }

   ~PersonInfo() // Destructor
     { delete [] name; }
```

```
// Accessor functions
   const char *getName()
     { return name; }

   int getAge()
     { return age; }

   // Overloaded = operator
   void operator=(const PersonInfo
     &right)
     { delete [] name;
       name = new
     char[strlen(right.name) + 1];
       strcpy(name, right.name);
       age = right.age; } };
```

```
PersonInfo person1("Molly McBride", 27);
PersonInfo person2 = person1;
```

# Returning a Value

- Overloaded operator can **return a value**

```
class Point2d
{
  public:
     double operator-(const point2d &right)
     { return sqrt(pow((x-right.x),2)
                 + pow((y-right.y),2)); }
...
  private:
     int x, y;
};
Point2d point1(2,2), point2(4,4);
// Compute & display distance between 2 points
cout << point2 - point1 << endl;
// displays 2.82843
```

# Returning a Value

- Return type the same as the left operand supports notation like:

```
object1 = object2 = object3;
```

- Function declared as follows:

```
const SomeClass operator=(const someClass &rval)
```

- In function, include as last statement:

```
return *this;
```

# Example

```
// Overloaded = operator
const PersonInfo PersonInfo::operator=(const PersonInfo
    &right)
{
    delete [] name;
      name = new char[strlen(right.name) + 1];
      strcpy(name, right.name);
      age = right.age;
    return *this;
}
```

```
PersonInfo person1("Molly McBride", 27);
PersonInfo person2, person3;
person3=person2=person1;
```

# The `this` Pointer

- **`this`**: predefined pointer available to a class's member functions
- **Always points to the instance (object)** of the class whose function is being called
- Can be used to access members that may be hidden by parameters with same name
- Is passed as a hidden argument to all **non-static member functions**

# Example: this Pointer

```
class SomeClass
{
   private:
         int num;
   public:
         void setNum(int num)
         { this->num = num; }
         ...
};
```

# Exercise

- Write definition of the 2 overloaded operator functions
- Write an appropriate main function to test the class.

```
class Rectangle {
   int height, width;
   public:
   Rectangle(int a=0,int b=0)
   {height=b; width=a;}
   int getWidth() { return width;}
   int getHeight() { return height;}
   friend Rectangle operator+(Rectangle,Rectangle);
   Rectangle operator-(Rectangle);
};
```

# Notes on Overloaded Operators

- Can change meaning of an operator
- Cannot change the number of operands of the operator
- Only certain operators can be overloaded. Cannot overload the following operators:

```
?:   .   .*   :: sizeof
```

# C++ operators that may be overloaded

| | | | | | |
|---|---|---|---|---|---|
| + | - | * | / | % | ^ |
| > | += | -= | *= | /= | %= |
| <<= | == | != | <= | >= | && |
| & | \| | ~ | ! | = | < |
| ^= | &= | \|= | << | >> | >>= |
| \|\| | ++ | -- | ->* | , | -> |
| [] | () | new | delete | | |

# Overloading Types of Operators

- ++, −− operators overloaded differently for prefix vs. postfix notation

- Overloaded relational operators should return a **`bool`** value

- Overloaded stream operators >>, << must **return reference to `istream`, `ostream` objects** and **take `istream`, `ostream` objects as parameters**

# Example: Relational Operators

```
class FeetInches {
private:
  int feet; int inches;
  void simplify();
public:
     FeetInches(int f = 0, int i = 0);
     void setFeet(int f);
     void setInches(int i);
     int getFeet() const;
     int getInches() const;
     FeetInches operator + (const FeetInches &); // Overloaded +
     FeetInches operator - (const FeetInches &); // Overloaded -
     FeetInches operator ++ ();   // Prefix ++
     FeetInches operator ++ (int);  // Postfix ++
     bool operator > (const FeetInches &);          // Overloaded >
     bool operator < (const FeetInches &);          // Overloaded <
     bool operator == (const FeetInches &);         // Overloaded ==
   friend ostream &operator << (ostream &, const FeetInches &);
   friend istream &operator >> (istream &, FeetInches &);
};
```

# Example: Prefix and Postfix

```
//Overloading prefix ++
FeetInches FeetInches::operator ++ ()
{
    ++inches;
    simplify();
    return *this;
}
//Overloading postfix ++
FeetInches FeetInches::operator ++ (int)
{
    FeetInches temp(feet, inches);
    inches++;
    simplify();
    return temp;
}
```

```
FeetInches first, second(1,5);

first=++second;
first=second++;
```

Dummy parameter

Copy to store data before increment

# Example: Relational Operator

```
bool FeetInches::operator > (const FeetInches &right){
    bool status;
    if (feet > right.feet)
            status = true;
    else if (feet == right.feet && inches > right.inches)
            status = true;
    else
            status = false;
    return status;
}
```

```
FeetInches first, second;
//setting first & second here
 if (first > second)
     cout << "first is greater than second.\n";
```

# Example: >> and <<

```
ostream &operator<<(ostream &strm, const FeetInches &obj){
  strm << obj.feet << " feet, " << obj.inches << " inches";
  return strm;
}
istream &operator >> (istream &strm, FeetInches &obj)
{  // Prompt the user for the feet.
     cout << "Feet: ";   strm >> obj.feet;
  // Prompt the user for the inches.
    cout << "Inches: ";   strm >> obj.inches;
  // Normalize the values.
    obj.simplify();
  return strm;
}
```

```
FeetInches first;
//setting first feet=6 Inches=5
cin>>first;
cout << first;
```

# Overloaded [ ] Operator

- Can create classes that behave like arrays, provide **bounds-checking** on subscripts
- Must consider constructor, destructor
- Overloaded [ ] returns a reference to object, not an object itself

# Object Conversion

# Object Conversion

- Can change meaning of an operatorType of an object can be converted to another type
- Automatically done for built-in data types
- Must write an operator function to perform conversion
- To **convert an `FeetInches`** object to an **`int`**:

```
FeetInches::operator int() {return feet;}
```

- Assuming distance is a `FeetInches` object, allows statements like:

```
FeetInches distance;
int d = distance;
```