UTM
UNIVERSITI TEKNOLOGI MALAYSIA

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

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
  …
};
```

Pointers to Objects

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;
rPtr->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.

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;
```

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
```

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

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 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;
}
```

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;
}
```

Operator Overloading

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

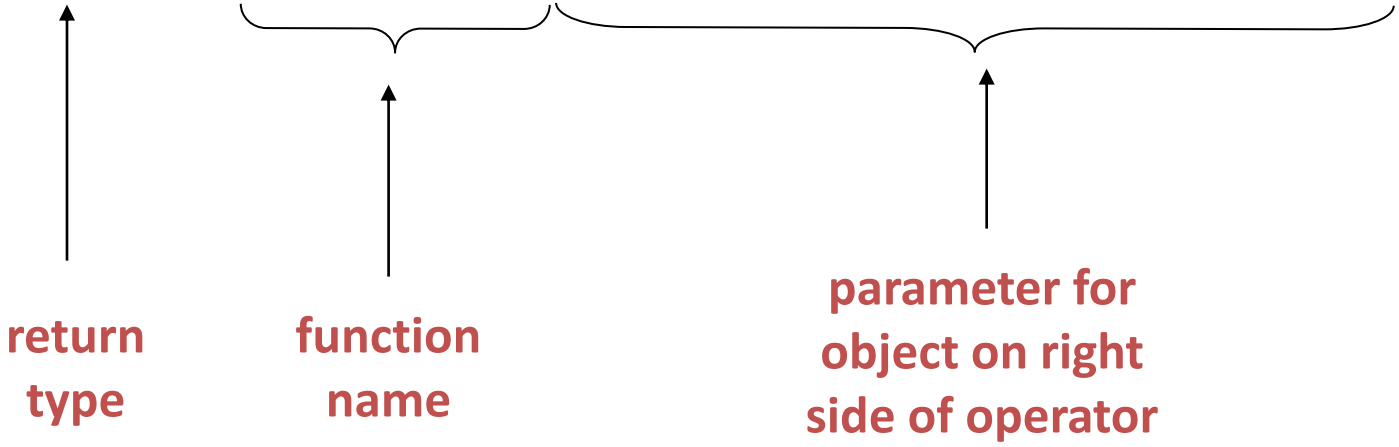
- Prototype for the overloaded operator goes in the declaration of the class that is overloading it

- 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**

Example:
`+ operator` with complex numbers

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

class TestClass {
private:
        int real, over;
public:
        TestClass(int rl = 0, int ov = 0) {
                real = rl;
                over = ov;
        }

        TestClass operator + (TestClass const &obj) {
                TestClass result;
                result.real = real + obj.real;
                result.over = over + obj.over;
                return result;
        }
        void print() {
                cout << real << " + " << over << endl;
        }
};
int main()
{
        TestClass c1(9, 5), c2(4, 3);
        TestClass c3 = c1 + c2;
        c3.print();
}
```

Using + operator for storing result in variable c3

Passing the values of both real and complex parts to be added.
The first part of c1 will be added to the first part of c2, that is, 9+4.
The second part of c1 will be added to the second part of c2, that is, 5+3.

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;
```

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;
}
```

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; }
         ...
};
```

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;
```