C++ slides - 5

Operator Overloading and Type Conversion: Syntax of operator overloading, Overloading Unary operator and Binary operator, Overloading arithmetic operator, relational operator, Overloading Unary operator and Binary operator using friend function, Data conversion, Overloading some special operators like (), []

Operator Overloading - definition

- These are the *member functions* to redefine the built-in operators with user-defined types.
- For example ‘+’ for string concatenation or adding dates of the format dd-mm-yyyy.

Ways to load an operator

Operator overloading can be achieved in two ways -

Member function OR non-member friend function.

Operator overloading – syntax

No return just increment e.g. ++obj;

```
void operator ++() {
//…
}
```

Return object after increment e.g. obj2 = ++obj1;

```
MyClass operator ++() {
//…
}
```

Operator overloading – syntax

For d3 = d1+d2; type operations in main

```
Distance operator+(Distance& d) {
Distance d3;
//.. set d3
return d3;
}
```

Unary operator

Operators that act upon a single operand to produce a new value. For example:

- unary minus(-)
- increment(++)

Overloading unary ‘-’ operator

```
Distance operator- () {
        feet = -feet;
        inches = -inches;
        return Distance(feet, inches);  // or return *this;
    }
```

Binary operator

- Operators that works with two operands are binary operators. For example:

a) Arithmatic (+ , – , * , /)
b) Relational (== or <= etc)
c) Logical (&& or || etc.)
d) Bitwise (&, | etc.)

Overloading binary ‘+’ operator

```
Complex operator + (Complex const &obj) {
        Complex c;
        c.real = real + obj.real;
        c.imag = imag + obj.imag;
        return c;
    }
```

```
class Complex {
    int real, imag;
public:
    Complex(int r = 0, int i =0)  {real = r;   imag = i;}
    Complex operator + (Complex const &obj) {
        Complex c;  c.real = real + obj.real;  c.imag = imag + obj.imag;
        return c;
    }
    void print() { cout << real << " + i" << imag << endl; }
};
int main() {
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2;  c3.print();
}
```

Write a function to overload Time object

Create a class Time with three private variables int h,m,s;

Create a function to overload + operator to add two time variables.

```
int main(){
    Time t1(5,15,34),t2(9,53,58),t3;
    t3 = t1 + t2;  t3.show();
}
```

Hint:

```
Time Time::operator+(Time t1) {
Time t;
int a,b;
a = s+t1.s;
t.s = a%60;     b =  (a/60)+m+t1.m;
t.m = b%60;  t.h = (b/60)+h+t1.h;
t.h = t.h%12;
return t;
}
```

Overloading *relational* operator

Example of overloading < operator in the Distance class

```
bool operator < (const Distance& d) {
        if(feet < d.feet) {
           return true;
        }
```

Overloading using friend function

- Friend functions allow to define operator overloading using non-member functions

```
class Test{
//…
public:
 friend void operator - (Test &x);
};
void operator-(Test &x){
//…
}
int main(){
Test x1;
-x1;
}
```

```
#include<iostream>
using namespace std;
class Test{
int i;
public:
Test(int a){i=a;}
void show(){ cout<<i<<endl;}
friend void operator - (Test &x);
};
void operator-(Test &x){ x.i = -x.i;}
int main(){
Test x1(11);
-x1; x1.show();
}
```

Data conversion

- Between basic types and user-defined types (UDT)
- Conversions between various user-defined types.

Basic to UDT

- Done by using the *constructor* with one argument of basic type as follows.

```
class Test {
private: //….
public:
Test ( data_type) { // conversion code }
};
```

```
class Cel{
float c;
public:
Cel(){c=0;}
Cel(float f){c=(f-32)* 5/9;}
void show(){cout<<"Celsius: "<<c;}
};
int main(){
    Cel cvalue(50);
    float f;
    cout<<“Fahrenheit : ";  cin>>f;
    cvalue=f;                    //conversion
    cvalue.show();
}
```

UDT to basic type

- Done by overloading the cast operator of basic type as a member function.

```
class Test{
 public:
operator data_type() { //Conversion code }
};
```

```
class Celsius{
float temper;
public:
//…
operator float(){
        float fer = temper *9/5 + 32;
        return fer;
    }
//…
};
int main(){
Celsius cel;        // finish code by setter & getter
float fer=cel;              // UDT to basic type
    cout<<fer;
}
```

One UDT to another UDT

- This conversion is exactly like conversion of UDT to basic type i.e. overloading the cast operator is used. For example

ClassA objA;

ClassB objB = objA;

So define the operator in the destination class ie. ClassB.

```
#include<math.h>
class Cartesian{
float xco, yco;
public:
Cartesian(float x=0, float y=0){ xco=x; yco=y;}
void display(){ cout<<xco<<yco;}
};
class Polar{
float r,a ;
public:
Polar(float r1=0, float a1=0){r =r1; a=a1;}
operator Cartesian(){
float x=r * cos(a); float y=r * sin(a);
return Cartesian(x,y);
   }
void display(){ cout<<r<<a;}
};
```

```
int main(){
    Polar pol(10.0, 0.78);
    Cartesian cart=pol;
    cart.display();
}
```

Overloading []

Following are some useful facts about overloading of []

- It is useful for index out of bound check.

- We must return reference in function because an expression like “arr[i]” can be used an lvalue. L-value means locator value which is an address in the memory.

```
class Array {
    int ptr[], size;
public:
   Array(int* p,int s) { ptr[0]=p[0]; ptr[1]=p[1];  size=s;}
  int& operator[](int i) {
    if (i >= size) {
        cout << "Array index out of bound, exiting"; exit(0);
    }
    return ptr[i];
}
};
int main() {
int a[] = { 1, 2 };
Array arr1(a,2);   arr1[8] = 6; }
```

Overloading ()

- Operator () can be overloaded for objects of class type.

- Overloading ( ) does not create function call but a function that can be accept arbitrary number of parameters.

```
class Distance {
  int feet,inches;
 public:
Distance(int f=0, int i=0) { feet = f; inches = i; }
Distance operator()(int a, int b) { //some params
return Distance(15,9);  //some value
}
void show() {cout << feet << " " << inches << endl;}   };
int main() {
   Distance D1, D2=D1(1,2);
   D1.show(); D2.show();
}
```