Operator Overloading

Operator Overloading

```
int a=5, b=10,c;
c = a + b;
```

Operator **+** performs **addition** of **integer operands** a, b

```
time t1,t2,t3;
t3 = t1 + t2;
```

Operator **+** performs **addition** of **objects** of type time

```
string str1=“Hello”
string str2=“Good Day”;
string str3;
str3 = str1 + str2;
```

Operator **+ concatenates** two strings str1,str2

Operator overloading

- **Function overloading** allow you to use same function name for different definition.
- **Operator overloading** extends the overloading concept to operators, letting you assign multiple meanings to C++ operators
- **Operator overloading** giving the normal C++ operators such as +, * and == additional meanings when they are applied with **user defined data types**.

Some of C++ Operators are already overloaded

Operator	Purpose
*	As pointer, As multiplication
<<	As insertion, As bitwise shift left
&	As reference, As bitwise AND

Operator Overloading

```
int a=5, b=10,c;
c = a + b;
```

Operator + performs addition of integer operands a, b

```
class time
{
  int hour, minute;
};

time t1,t2,t3;
t3 = t1 + t2;
```

Operator + performs addition of objects of type time t1,t2

```
string str1=“Hello”,str2=“Good Day”;
str1 + str2;
```

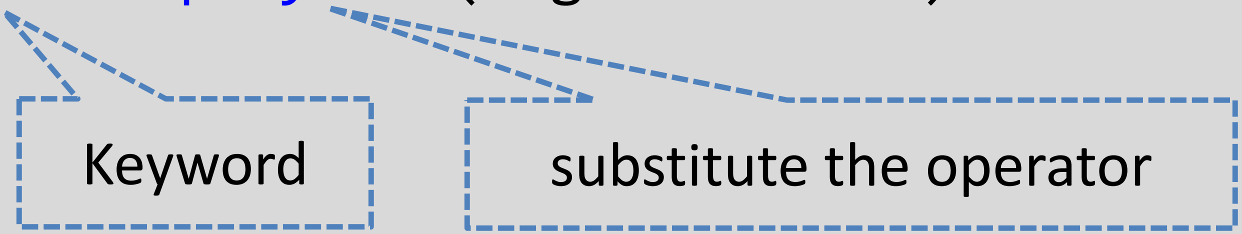
Operator + concatenates two strings str1,str2

Operator Overloading

- Specifying more than one definition for an **operator** in the same scope, is called **operator overloading**.
- You can overload operators by creating ***“operator functions”***.

Syntax:

```
return-type operator op-symbol(argument-list)
{
   // statements
}
```



Keyword

substitute the operator

Example:

```
void operator + (arguments);
int operator - (arguments);
class-name operator / (arguments);
float operator * (arguments);
```

Overloading Binary operator +

```
class complex{
  int real,imag;
  public:
    complex(){
     real=0; imag=0;
    }
    complex(int x,int y){
     real=x; imag=y;
    }
    void disp(){
     cout<<"\nreal value="<<real<<endl;
     cout<<"imag value="<<imag<<endl;
    }
    complex operator + (complex);
};
complex complex::operator + (complex c){
  complex tmp;
  tmp.real = real + c.real;
  tmp.imag = imag + c.imag;
  return tmp;
}
```

```
int main()
{
  complex c1(4,6),c2(7,9);
  complex c3;
  c3 = c1 + c2;
  c1.disp();
  c2.disp();
  c3.disp();
  return 0;
}
```

Similar to function call

`c3=c1.operator +(c2);`

Binary Operator Arguments

```
result = obj1.operator symbol (obj2);//function notation
```

```
result = obj1 symbol obj2;           //operator notation
```

```
complex operator + (complex x)
{
   complex tmp;
   tmp.real = real + x.real;
   tmp.imag = imag + x.imag;
   return tmp;
}
```

```
result = obj1.display();
```

```
void display()
{
   cout<<"Real="<<real;
   cout<<"Imaginary="<<imag;
}
```

Operator Overloading

- **Operator overloading** is compile time polymorphism.
- You can overload most of the built-in operators available in C++.

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

Operator Overloading using Friend Function

Invoke Friend Function in operator overloading

```
result = operator symbol (obj1,obj2);//function notation
```

```
result = obj1 symbol obj2;           //operator notation
```

```
friend complex operator +(complex c1,complex c2)
{
  complex tmp;
  tmp.r=c1.r+c2.r;
  tmp.i=c1.i+c2.i;
  return tmp;
}
```

```
int main()
{
  complex c1(4,7),c2(5,8);
  complex c3;
  c3 = c1 + c2;
  c3 = operator +(c1,c2);
}
```

Overloading Binary operator ==

```
class complex{
  int r,i;
  public:
  complex(){
    r=i=0;}
  complex(int x,int y){
    r=x;
    i=y;}
  void display(){
   cout<<"\nreal="<<r<<endl;
   cout<<"imag="<<i<<endl;}
   int operator==(complex);
};
int complex::operator ==(complex c){
  if(r==c.r && i==c.i)
    return 1;
  else
    return 0;}
```

```
int main()
{
  complex c1(5,3),c2(5,3);
  if(c1==c2)
    cout<<"objects are equal";
  else
    cout<<"objects are not equal";
    return 0;
      }
```

Overloading Unary Operator

Overloading Unary operator –

```
class space {
  int x,y,z;
  public:
  space(){
    x=y=z=0;}
  space(int a, int b,int c){
    x=a; y=b; z=c; }
  void display(){
   cout<<"\nx="<<x<<",y="<<y<<",z="<<z;
  }
  void operator-();
};
void space::operator-() {
  x=-x;
  y=-y;
  z=-z;
}
```

```
int main()
{
  space s1(5,4,3);
  s1.display();
  -s1;
  s1.display();
  return 0;
        }
```

Overloading Unary operator ––

```
class space {
  int x,y,z;
  public:
  space(){
    x=y=z=0;}
  space(int a, int b,int c){
    x=a; y=b; z=c; }
  void display(){
   cout<<"\nx="<<x<<",y="<<y<<",z="<<z;
  }
  void operator--();
};
void space::operator--() {
  x--;
  y--;
  z--;
}
```

```
int main()
{
  space s1(5,4,3);
  s1.display();
  --s1;
  s1.display();
  return 0;
        }
```

Overloading Prefix and Postfix operator

```
class demo
{
    int m;
    public:
     demo(){ m = 0;}
     demo(int x)
     {
       m = x;
     }
     void operator ++()
     {
        ++m;
        cout<<"Pre Increment="<<m;
     }
     void operator ++(int)
     {
        m++;
        cout<<"Post Increment="<<m;
     }
};
```

```
int main()
{
    demo d1(5);
    ++d1;
    d1++;
}
```

Invoking Operator Function

- Binary operator

```
operand1 symbol operand2
```

- Unary operator

```
operand symbol
symbol operand
```

- Binary operator using friend function

```
operator symbol (operand1,operand2)
```

- Unary operator using friend function

```
operator symbol (operand)
```

Rules for operator overloading

- Only existing operator can be overloaded.
- The overloaded operator must have at least one operand that is user defined type.
- We cannot change the basic meaning and syntax of an operator.

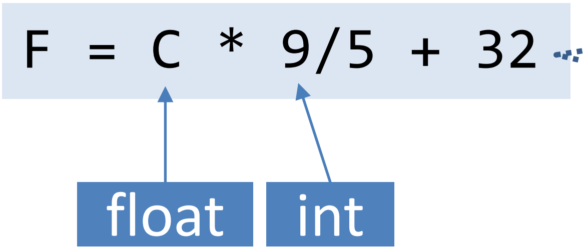
Rules for operator overloading (Cont…)

- When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.
- We cannot overload following operators.

Operator	**Name**
. and .*	Class member access operator
::	Scope Resolution Operator
sizeof()	Size Operator
?:	Conditional Operator

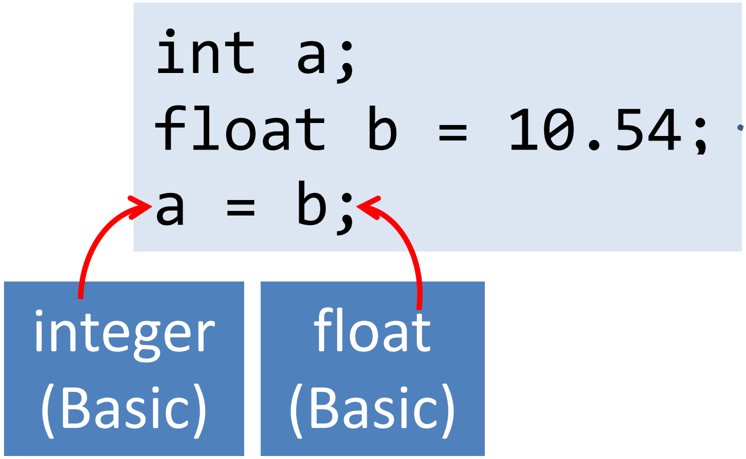
Type Conversion

Type Conversion



```
F = C * 9/5 + 32
```

If different data types are mixed in expression, C++ applies automatic type conversion as per certain rules.



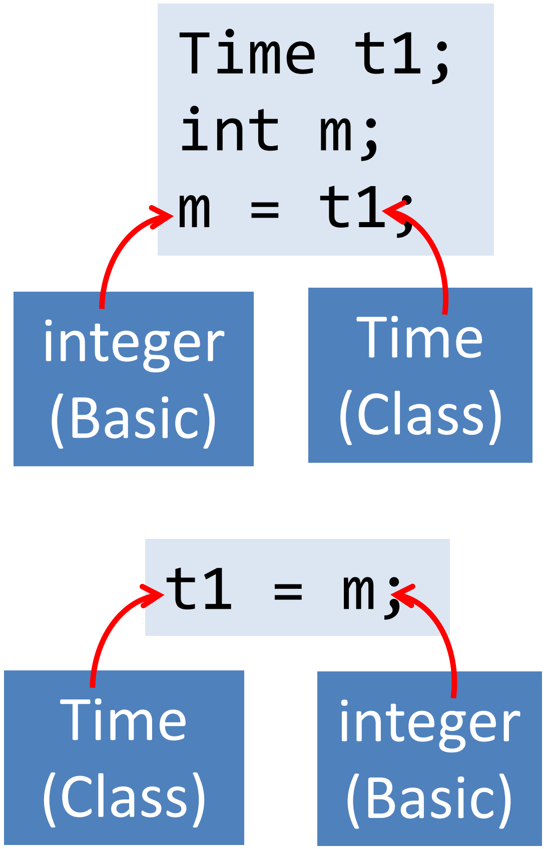
```
int a;
float b = 10.54;
a = b;
```

a = 10;

- float is converted to integer automatically by complier.
- basic to basic type conversion.

- An assignment operator causes automatic type conversion.
- The data type to the right side of assignment operator is automatically converted data type of the variable on the left.

Type Conversion



```
Time t1;
int m;
m = t1;
```

```
t1 = m;
```

- class type will not be converted to basic type OR basic type will not be converted class type automatically.

Type Conversion

- C++ provides mechanism to perform automatic type conversion if all variable are of **basic type**.
- For user defined data type programmers have to convert it by using **constructor** or by using **casting operator**.

- Three type of situation arise in user defined data type conversion.
  1. Basic type to Class type (Using Constructors)
  2. Class type to Basic type (Using Casting Operator Function)
  3. Class type to Class type (Using Constructors & Casting Operator Functions)

(1) Basic to class type conversion

- Basic to class type can be achieved **using constructor**.

```
class sample
{
  int a;
  public:
  sample(){}
  sample(int x){
    a=x;
  }
  void disp(){
    cout<<"The value of a="<<a;
  }
};
```

```
int main()
{
  int m=10;
  sample s;
  s = m;
  s.disp();
  return 0;
}
```

(2) Class to basic type conversion

- The Class type to Basic type conversion is done **using casting operator function**.
- The casting operator function should satisfy the following conditions.
  1. It must be a class member.
  2. It must not mention a return type.
  3. It must not have any arguments.

Syntax:

```
operator destinationtype()
{
    ....
    return
}
```

Program: Class to basic type conversion

```
class sample
{
   float a;
   public:
   sample()
   {
     a=10.23;
   }
   operator int() //Casting operator
                    function
   {
     int x;
     x=a;
     return x;
   }
};
```

```
int main()
{
  sample S;
  int y= S;//Class to Basic
              conversion
  cout<<"The value of y="<<y;
  return 0;
}
```

Explicit type conversion

`y = int (S);`

Automatic type conversion

`y = S;`

Program: Class to basic type conversion

```
class vector{
   int a[5];
   public:
   vector(){
      for(int i=0;i<5;i++)
            a[i] = i*2;
   }
   operator int();
};
vector:: operator int() {
   int sum=0;
   for(int i=0;i<5;i++)
      sum = sum + a[i];
   return sum;}
```

```
 int main()
{
vector v;
int len;
len = v;
cout<<“Length of V="<<len;
return 0;
}
```

(3) Class type to Class type

- It can be achieved by two ways
    1. Using constructor
    2. Using casting operator function

Program: Class type to Class type

```
class alpha
{
  int commona;
  public:
    alpha(){}
    alpha(int x)
    {
      commona = x;
    }
    int getvalue()
    {
      return commona;
    }
};
```

```
int main()
{
  alpha obja(10);
  beta objb(obja);
  beta objb(20);
  obja = objb;
 }
```

```
class beta
{
   int commonb;
   public:
     beta(){}
     beta(int x)
     {
       commonb = x;
     }
     beta(alpha temp)  //Constructor
     {
       commonb = temp.getvalue();
     }
     operator alpha()  //operator function
     {
       return alpha(commonb);
     }
};
```

Program: Type Conversion

```
class stock2 ;
class stock1{
  int code , item ;
  float price ;
  public :
  stock1 ( int a , int b , int c ) {
   code = a ; item = b ; price = c ;
  }
  void disp ()   {
   cout << " code " << code << " \n " ;
   cout << " items " << item << " \n " ;
   cout << " price per item Rs. " << price << " \n " ;
  }
  int getcode (){ return code; }
  int getitem (){ return item ; }
  int getprice (){ return price ; }
  operator float ()  {
   return ( item*price ) ;
  }
};
```

Program: Type Conversion

```
class stock2{
  int code ;
  float val ;
  public :
  stock2 ()  {
    code = 0; val = 0 ;
  }
  stock2( int x , float y ){
    code = x ; val = y ;
  }
  void disp ()  {
   cout << " code " << code << " \n " ;
   cout << " total value Rs. " << val << " \n " ;
  }
  stock2( stock1 p )  {
    code = p.getcode() ;
    val = p.getitem() * p.getprice() ;
  }
};
```

Program: Type Conversion

```
int main()
{
  stock1 i1 ( 101 , 10 ,125.0 ) ;
  stock2 i2 ;
  float tot_val = i1;
  i2 = i1 ;
  cout << " Stock Details : Stock 1 type " << " \n " ;
  i1.disp ();
  cout << " Stock Value " << " - " ;
  cout << tot_val << " \n " ;
  cout << " Stock Details : Stock 2 type " << " \n " ;
  i2.disp () ;
  return 0 ;
}
```

Thank You