Day: Date: / /

Operator overloading:

- Consider member-wise assignment:

```
E2 = E1;
```

how does "=" operator know to copy every thing from E1 to E2.

- This is done by operator overloading.

Operators that can be overloaded:

1) +	7) &&	13) +=	19) new	
2) -	8) |	14) *=	20) delete.	
3) *	9) &&	15) <<		
4) \	10) ||	16) >>		
5) %	11) !	17) >>=		
6) ^	12) ~	18) -<		

These are just a few examples.

Operators that CANNOT be overloaded:

1) ?:
2) ::
3) .
4) sizeof.
5) .*

Yousaf

Day: Date: / /

A few rules:

1) Built-in behaviour cannot change.
e.g

```
int x=5;
x = x/15;
```
← This will return an integer and cannot change this behaviour.

2) Precedence and associativity cannot change

3) Ternary cannot be overloaded as only one ternary (?:)

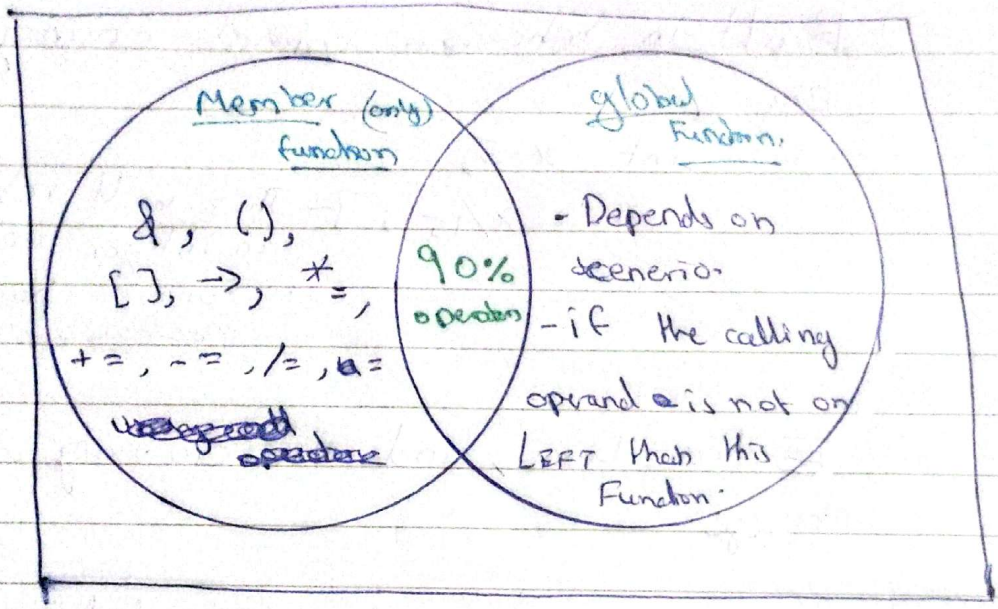
4) Binary cannot become unary operator and vice versa.

5) But some operator can do both
e.g +, -, *, &.

Yousaf

Day: Date: / /

Types of operator functions:



~~Overloading Class operator~~

~~overloading~~

Syntax

keyword. e.g =, &, ().

```
return type  operator operator (Parameters) {
    body.
}
```

Yousaf

Class that will be used in this section:

```
class Complex{
Private:
    int real;
    int img;

Public:

    Complex(int r=0, int i=0) : real(r), img(i)
    { }; //construtor

    Complex (const iComplex & c): real(c.real),
    img(c.img) { }; //copy constructor

    int getR() const { return real; }
    int getI() const { return img; }
    void setR(int r){ real = r; }
    void setI(int i) { img = i; }

    void display() {
        cout << real << ", " << img << i << endl;
    }
```

All other function will be automatically made in each operator.

Reg: Date: / /

1) "=" operator:

- This is a binary operator.
- This can ONLY be a member function.

Function inside complex Class:

```
void operator = (const complex &c){
    real = c.real;
    img = c.img;
}
```

An important point about member function:
consider member operators like.
object.operator(value)
e.g
E1 = E2 mean.
E1.operator(E2).

But there is problem what if in main it was written:

c = c1 = c2;

This will return nothing

So this be nothing.

Yousaf

So correct function:

```
Complex operator = (const complex & c) {

    real = c.real;
    img = c.img;

    Complex temp ( real, img);

    return temp;
}
```

```
int main() {

    Complex c(2, 3);

    Complex c1, c2;

    c2 = c1 = c;

    c2.display();
}
```

This means:

c2.operator(c1.operator(c))

Terminal:

```
2.3i
```

Day: Date: / /

2) " + " operator as a binary operator.

eg in main it is written:
C + 2;
, now this 2 should be added in real part of C.

If we consider C+2, we can consider it like:

C. operator (2);

This shows one parameter.

- This should also return a temp as we could also have a case of like C+2+3+4;

Member function inside Complex class:

```
Complex operator + (int r) {
    Complex temp( this->real+r , img);
    return temp;
}
```

- In this case we are ~~only~~ ~~increasing~~ not changing C as C+2 doesn't change it but C=C+2 will change it.

Yousaf

Page: Date: / /

2) another case:

what if we treat:

C + C1 + C2.

how we can consider it as:

C.operator (complex (1).

```
Complex operator + ( complex & c){
    Complex temp(real + c.real, img + c.img);
    return temp;
}
```

Now we can write

C + 1 + C1 + C2 + 2;

Day: Date: / /

3) Another case: (Imp)

what if we had written:

2 + c2;

This means

2 . operator (c2).

but the thing 2 is not an object.

- So now we require a global operator function.

Global ~~opp~~ functions:

- In global functions operator is the caller.
- In member functions left operand was the caller.
- So now in global functions there will be 2 parameters.

Yousaf

Day: Date: / /

```
Complex operator + ( int r, const Complex &C){
        Complex temp (r+c.getR, c.getI);
        return temp;
    }
```

global function.

- But now we have written 3 function to do the work.
- what if we use only one Global function.

```
Complex operator + (Complex c1, Complex c2){
    Complex temp (c1.getR + c2.getR ,
                  c1.getI + c2.getI);
    return temp;
}
```

only one for all the stuff.

always include one user defined data type in parameters or else error e.g

```
Complex operator +
(int r, int j) { };
```

✗ error

but

```
Complex operator +
(int r, Example E)
{ }.
```

✓ allowed!

In this we used pass by value because e.g for c1 + 2 - 2 is an integer and when integer is send it will be changed to complex by constructor.

Yousaf

Day: Date: / /

4) Another case:

What if we had:

+C1;

a unary operator defining (-) as +ve.

This mean C1.operator();

- Now this can ONLY be member, not global because global requires parameters

Member function:

```
Complex operator+ () {
    complex temp (real, img);
    return temp;
}
```

- But there is a way to make it global:

Date: / /

if + becomes caller and if we have one parameter then this will work:

```
Complex operator + ( complex & c) {
        Complex temp( c.real, c.img);
        return temp.
}
```

global.

Day: Date: / /

3) "-", "*", "/" operators.

same as + operator.

4) "++" prefix and postfix.

They work like unary operators but one comes before and one comes after.

Prefix:

- changes value and returns the same object.

so code:

member:

```
Complex & operator ++ ( ) {
    (*this) = (*this) + 1; // using the + operator code already written.
    return *this;
}
```

global:

```
Complex & operator ++ (Complex &C) {
    C = C+1;
    return C;
}
```

Yousaf

Day: Date: / /

Post fix:

This returns the original and incraments the original by one.

- So a value is returned.

- To differentiate a postfix and prefix we include a dummy variable inside the parameter which represents a postfix

This dummy var can be considered as symbol for postfix.

dummy variable.

Code:

Member:

```
Complex operator ++ (int dummy) {
    Complex temp = *this;
    (*this) = (*this) + 1;
    return temp;
}
```

global:

```
Complex operator ++ ( Complex & C, int dummy) {
    Complex temp = C;
    C = C+1;
    return temp;
}
```

Yousaf

Prog. Date: / /

3) " [ ] " array brackets;

with these we can apply and find the object at perticular index of the object array.

• • [ ] can only be Member function

Example using a String class:

a string class has variables as a Char pointer and a size.

now to access the char array we can use the [] operator.

an address is returned.

```
Char & operator [ ] (int index) {
    if ((index >= 0) && (index < size)) {
        return charArray[index];
    }
}
```

Yousaf

Day:

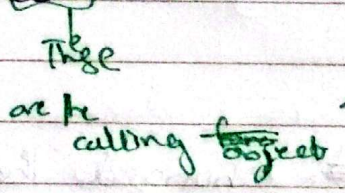
Date: / /

(6) "<<" and ">>" operators.

- These are normally used for outputs and inputs ie cout and cin.

- cout and cin are themselves objects coming from iostream Library.

- if we are making << and >> for cout, cin these will be global functions as cout << ---, cin << ---



- These functions should return their particular cout or cin meaining ostream or istream.

- Also remember ostream and istream can only be passed by reference. as their copy constructor is unusable due to being deleted.

Example of cout as a global function

```
1  ostream & operator << (ostream & out , Complex &C){
2
3          out << "( " << C.getR() << "+" << C.getI() << "i)";
4
5          return out;
6
7  }
```

In line 1:

I changed the name of cout to only out to differentiate it from the global cout.

In line 3:

- I added no endl, because that is something we normally add in the main

In line 5:

- we need to return ostream object bcz. e.g cout << C1 << endl;

This should return cout so we could use it.

Yousaf

Page: Date: / /

Example of cin:

```
1   istream & operator >> (istream & in, Cplex & C) {
2           int t; cout << "Enter Real: ";
3           in >> t;
4           C.setR(t);
5           cout << " Enter Imaginery: ";
6           in >> t;
7           C.setI(t);
8
9           return in;
10  }
```

In line 1:

- istream is for cin while
- ostream is for cout.

Yousaf

7) Casting operators : very imp

We know about casting e.g (double) b; (int b;)
This makes b a double.

- To do this we also have an operator.

```
operator. Datatype (●) {

        body;
}
```

- This is always Inline.
- This has NO return type.

- Datatypes can be USER-defined datatypes.
  i.e own created classes.

- Example next page.

- Return type is the Datatype.

Day: Date: / /

Example using Complex class and another Point class.

```
Class Point {

    int x;
    int y;

  Public:
      Point (int x, int y) : x(x), y(y) { } // constructor
      operator Complex () { // casting.

          return Complex(x,y);

      }
};
```

In the above class we are making the Point into complex. Remember. Here both are User defined. It is also possible for double (), int() anything.

- The return type is the thing we are casting into e.g in this it is complex().

Yousaf

Day: Date: / /

```
int main () {

    Point ( 3, 5);

    cout << ( (Complex) Point ) << endl;

};
```

Terminal:

(3 + 5i)

- we had already created the cout << function for Complex which were used as Point was casted into complex.

Yousaf

Day: Date: / /

8) " ( ) " operators

- This can be inline and global.

Syntax:

```
Return type  operator ( ) (parameters) {
                body;
             }
```

example using Complex class. (inline functions).

```
Complex & operator () () {
```

don't be confused one is to tell which operator and other for parameters.

```
    return *this;
}.
```

```
Complex & operator () (int x, int y) {

    real = x;
    Imag = y;

    return this;
}.
```