

CHAPTER-6

Introducing Operator Overloading

The Basics of Operator Overloading:

- Operator overloading is really just a type of function overloading. However some additional rules apply. An operator is always overloaded relative to a user defined type, such as a class.
- When an operator is overloaded, that operator loses none of its original meaning. Instead, it gains additional meaning relative to the class for which it is defined.
- To overload an operator, you create an *operator function*. Most often an operator function is a member or a friend of the class for which it is defined. However, there is a slight difference between a member operator function and a friend operator function.

The general form of a member operator function:

return-type class-name::operator#(arg-list)

{

//operation to be performed }

The operator being overloaded is substituted for the #

The operators that you can not overload are shown here:

. :: .* ?:

- The precedence of the operator cannot be changed.
- The number of operands that an operator takes cannot be altered.
- Except for the "=" operator functions are inherited by any derived classes. However, a derived class is free to overload any operator it chooses.
- While it is permissible for you to have an operator function perform any activity whether related to the traditional use of the operator or not, it is best to have an overloaded operator's actions stay within the spirit of the operator's traditional use.

Overloading Binary Operators

When a member operator function overloads a binary operator, the function will have only one parameter. This parameter will receive the object that is on the right side of the operator. The object on the left side is the object that generates the call to the operator function and is passed implicitly by **this**.

Example:

```
#include <iostream>
using namespace std;
class coord {private:
int x,y;
public:
coord () {x=0;y=0;}
coord (int i,int j) {x=i;y=j;}
void get_xy(int &i,int &j) {i=x;j=y;}
coord operator+(coord ob2);
};
```

```
coord coord::operator+(coord ob2)
{
    coord temp;
    temp.x=x+ob2.x;// temp.x=this->x+ob2.x;
    temp.y=y+ob2.y;// temp.y=this->y+ob2.y;
    return temp;
}
int main() {
    coord o1(10,10),o2(5,3),o3,o4;
    int x,y;
    o3=o1+o2; //o3=o1.operator+(o2);
    o3.get_xy(x,y);
    cout<<"(o1+o2) X:"<<x<<", Y:"<<y<<endl;
    o4=o1+o2+o3;//This statement is valid
    return 0;
}
```

Note: When a binary operator is overloaded, the left operand is passed implicitly to the function and the right operand is passed as an argument.

Example:

```
#include <iostream>
using namespace std;
class coord {private:
int x,y;
public:
coord () {x=0;y=0;}
coord (int i,int j) {x=i;y=j;}
void get_xy(int &i,int &j) {i=x;j=y;}
coord operator+(coord ob2);
coord operator-(coord ob2);
coord operator=(coord ob2);
};
coord coord::operator+(coord ob2)
{coord temp;
temp.x=x+ob2.x;
temp.y=y+ob2.y;
return temp;}
```

```
coord coord::operator-(coord ob2)
{
    coord temp;
    temp.x=x-ob2.x;
    temp.y=y-ob2.y;
    return temp;
}

coord coord::operator=(coord ob2)
{
    x=ob2.x;
    y=ob2.y;
    return *this; //return the object that is assigned.
}
```

```
int main() {  
coord o1(10,10),o2(5,3),o3;  
int x,y;  
o3=o1+o2; //o3=o1.operator+(o2);  
o3.get_xy(x,y);  
cout<<"(o1+o2) X:"<<x<<", Y:"<<y<<endl;
```

```
o3=o1-o2; //o3=o1.operator-(o2);  
o3.get_xy(x,y);  
cout<<"(o1-o2) X:"<<x<<", Y:"<<y<<endl;
```

```
o3=o1; //o3.operator=(o1); assign an object.  
o3.get_xy(x,y);  
cout<<"(o3=o1) X:"<<x<<", Y:"<<y<<endl;  
o3=o2=o1;//this is also a valid assignment.  
return 0;  
}
```

Example: overload ‘+’ for object+int as well as object+object.

```
#include <iostream>
using namespace std;
class coord {private:
    int x,y;
public:
    coord () {x=0;y=0;}
    coord (int i,int j) {x=i;y=j;}
    void get_xy(int &i,int &j) {i=x;j=y;}
    coord operator+(coord ob2);
    coord operator+(int i);
};
coord coord::operator+(int i)
{coord temp;
temp.x=x+i;
temp.y=y+i;
return temp;}
```

```
coord coord::operator+(coord ob2){  
    coord temp;  
    temp.x=x+ob2.x;  
    temp.y=y+ob2.y;  
    return temp;  
}  
int main()  
{  
    coord o1(10,10),o2(5,3),o3;  
    int x,y;  
    o3=o1+o2; //o3=o1.operator+(o2);  
    o3.get_xy(x,y);  
    cout<<"(o1+o2) X:<<x<<", Y:<<y<<endl;  
    o3=o1+100; //o3=o1.operator+(100);  
    o3.get_xy(x,y);  
    cout<<"(o1+100) X:<<x<<", Y:<<y<<endl;  
    return 0;  
}  
o3=100+o1; //we cannot do this
```

You can use a reference parameter in an operator function.

```
coord coord::operator+(coord &ob2)
{
    coord temp;
    temp.x=x+ob2.x;
    temp.y=y+ob2.y;
    return temp;
}
```

Overloading the Relational and Logical Operators

When you overload the relational and logical operators so that they behave in their traditional manner, you will not want the operator functions to return an object of the class for which they are defined. Instead, they will return an integer that indicates either true or false.

Örnek:

```
#include <iostream>
using namespace std;
class coord { private: int x,y;
public:
coord () {x=0;y=0;}
coord (int i,int j) {x=i;y=j;}
void get_xy(int &i,int &j) {i=x;j=y;}
int operator==(coord ob2);
int operator&&(coord ob2);
};
```

```
int coord::operator==(coord ob2) {return (x==ob2.x) && (y==ob2.y);}
int coord::operator&&(coord ob2){return (x&&ob2.x) && (y&&ob2.y);}
void f1(coord o1,coord o2,char *x,char *y)
{
if (o1==o2) cout<<x<<" and "<<y<<" are same"<<endl;
else cout<<x<<" and "<<y<<" are different"<<endl;}
void f2(coord o1,coord o2,char *x,char *y)
{
if (o1&&o2) cout<<x<<" && "<<y<<" is true"<<endl;
else cout<<x<<" && "<<y<<" is false"<<endl;
}
int main()  {setlocale(LC_ALL,"Turkish");
coord o1(10,20), o2(5,3), o3(10,20), o4(0,10);
f1(o1,o2,"o1","o2");
f1(o1,o3,"o1","o3");
f2(o1,o2,"o1","o2");
f2(o1,o4,"o1","o4");
return 0;}
```

Overloading a Unary Operator

When you overload a unary operator using a member function, the function has no parameters. Since there is only one operand, it is this operand that generates the call to the operator function.

Example:

```
#include <iostream>
using namespace std;
class coord {private:
int x,y;
public:
coord () {x=0;y=0;}
coord (int i,int j) {x=i;y=j;}
void get_xy(int &i,int &j) {i=x;j=y;}
coord operator++();
};
```

```
coord coord::operator++()
{
    x++;y++;
    return *this;
}
int main()
{
    coord o1(10,10);
    int x,y;
    ++o1; //Increment an object.
    o1.get_xy(x,y);
    cout<<"(++o1) X:"<<x<<", Y:"<<y<<endl;
    return 0;
}
```

There are two versions of increment(++) and decrement (--) operators in C++. a) prefix increment (decrement), b) postfix increment (decrement)

Example:

```
int x,y;  
x=0;  
y=x++;  
//postfix increment. x will be 1 and y will be 0 after executing the following  
//line.  
x=5;  
y=++x;  
//prefix increment. x will be 1 and y will be 1 after executing the following  
//line.
```

Prefix and postfix versions of ++ and – operators can be overloaded for a given class.

Declaration of prefix version of ++ (--) operator :

```
class-name class-name::operator++();
```

Declaration of postfix version of ++ (--) operator :

```
class-name class-name::operator++(int notused);
```

Example: prefix, postfix increments

```
#include <iostream>
using namespace std;
class coord {private:
int x,y;
public:
coord () {x=0;y=0;}
coord (int i,int j) {x=i;y=j;}
void get_xy(int &i,int &j) {i=x;j=y;}
coord operator++(); //prefix increment
coord operator++(int notused); //postfix increment
coord operator=(coord ob2);
};
coord coord::operator=(coord ob2)
{
x=ob2.x;
y=ob2.y;
return *this; //return the object that generates the call.
}
```

```
coord coord::operator++(){x++;y++; return *this;}
coord coord::operator++(int notused) { coord temp;
temp.x=x
temp.y=y;
x++;y++;
return temp;}
int main(){coord o1(10,10),o2; int x,y;
o2=++o1; //prefix increment.
o1.get_xy(x,y);
cout<<"(++o1) X:"<<x<<", Y:"<<y<<endl;
o2.get_xy(x,y);
cout<<"(o2) X:"<<x<<", Y:"<<y<<endl;
o2=o1++; //postfix increment.
o1.get_xy(x,y);
cout<<"(o1++) X:"<<x<<", Y:"<<y<<endl;
o2.get_xy(x,y);
cout<<"(o2) X:"<<x<<", Y:"<<y<<endl;
return 0;}//Burada kaldık.09-12-2013
```

-sign is both a binary an unary operator in C++. We can overload – sign twice, once as a binary operator and once as a unary operator.

Example:

```
#include <iostream>
using namespace std;
class coord {private: int x,y;
public:
coord () {x=0;y=0;}
coord (int i,int j) {x=i;y=j;}
void get_xy(int &i,int &j) {i=x;j=y;}
coord operator-(); //Unary operator
coord operator-(coord ob2); //Binary operator
};
coord coord::operator-(coord ob2) //Binary operator
{ coord temp;
temp.x=x-ob2.x;
temp.y=y-ob2.y;
return temp;}
```

```
coord coord::operator-() //Unary operator
{x=-x;y=-y; return *this;}
int main()
{
coord o1(10,10),o2(5,7);
int x,y;
o1=o1-o2; //Substraction
o1.get_xy(x,y);
cout<<"(o1-o2) X:"<<x<<", Y:"<<y<<endl;
o1=-o1; //Negation
o1.get_xy(x,y);
cout<<"(-o1) X:"<<x<<", Y:"<<y<<endl;
return 0; }
```

Using Friend Operator Functions

It is possible to overload an operator relative to a class by using a friend rather than a member function.

Example:

```
#include <iostream>
using namespace std;
class coord {private: int x,y;
public:
coord () {x=0;y=0;}
coord (int i,int j) {x=i;y=j;}
void get_xy(int &i,int &j) {i=x;j=y;}
friend coord operator+(coord ob1, coord ob2);
};
```

```
//overload + using a friend function.  
coord operator+(coord ob1, coord ob2);  
{  
coord temp;  
temp.x=ob1.x+ob2.x;  
temp.y=ob1.y+ob2.y;  
return temp;  
}  
int main()  
{  
coord o1(10,10),o2(5,3),ob3;  
int x,y;  
o3=o1+o2; //add two object  
o3.get_xy(x,y);  
cout<<"(o1+o2) X:"<<x<<", Y:"<<y<<endl;  
return 0;  
}
```

Notice that the left operand is passed to the first parameter and the right operand is passed to the second parameter.

Using a friend operator function, you can allow objects to be used in operations involving built-in types in which the built-in type is on the left side of the operator. (example: ob1=10+ob2;)

Example:

```
#include <iostream>
using namespace std;
class coord {private: int x,y;
public:
coord () {x=0;y=0;}
coord (int i,int j) {x=i;y=j;}
void get_xy(int &i,int &j) {i=x;j=y;}
friend coord operator+(coord ob1, int i);
friend coord operator+(int i,coord ob1);
};
coord operator+(coord ob1, int i) {
coord temp;
temp.x=ob1.x+i;
temp.y=ob1.y+i;
return temp;}
```

```
coord operator+(int i, coord ob1)
{
    coord temp;
    temp.x=ob1.x+i;
    temp.y=ob1.y+i;
    return temp;
}
int main()
{
    coord o1(10,10);
    int x,y;
    o1=o1+10; //object + integer
    o1.get_xy(x,y);
    cout<<"(o1+10) X:"<<x<<", Y:"<<y<<endl;
    o1=99+o1; // integer + object
    o1.get_xy(x,y);
    cout<<"(99+o1) X:"<<x<<", Y:"<<y<<endl;
    return 0;
}
```

If you want to use a friend function to overload either the **++** or **-** unary operator, you must pass the operand to the function as a reference parameter.

Example:

```
#include <iostream>
using namespace std;
class coord {
private: int x,y;
public:
coord () {x=0;y=0;}
coord (int i,int j) {x=i;y=j;}
void get_xy(int &i,int &j) {i=x;j=y;}
friend coord operator++(coord &ob);
};
```

```
coord operator++(coord &ob)
{
    ob.x++;
    ob.y++;
    return ob; //return object generating the call
}
int main()
{
    coord o1(10,10);
    int x,y;
    ++o1; //o1 is passed by reference.
    o1.get_xy(x,y);
    cout<<"(++o1) X:"<<x<<", Y:"<<y<<endl;
    return 0;
}
```

Here are the prototypes for both the prefix and postfix versions of the increment operator relative to the **coord** class.

```
coord operator++(coord &ob) ;//prefix
```

```
coord operator++(coord &ob, int notused) ; //postfix
```

A Closer Look at the Assignment Operator

By default, when the assignment operator is applied to an object, a bitwise copy of the object on the right is put into the object on the left. However, there are cases in which a strict bitwise copy is not desirable. You do not want a bitwise copy of the object when the object allocates memory. In these types of situations, you will want to provide a special assignment operation.

Example:

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;
class strtype {private: char *p;int len;
public:
strtype(char *s);//constructor
~strtype() {delete []p;}//destructor
char *get() {return p;}
strtype &operator=(strtype &ob);
};
```

```
strtype::strtype(char *s){int l;
l=strlen(s)+1;
p=new char [l];
if (!p) {cout<<"Allocation error"<<endl; exit (1);}
strcpy(p,s);len=l;
}
strtype &operator=(strtype &ob) //assign an object
if (len<ob.len) {delete [] p; p=new char [ob.len];
if (!p) {cout<<"Allocation error"<<endl; exit(1);}
}
len=ob.len; strcpy (p,ob.p);
return *this;}
int main(){ strtype a("Hello"),b("There");
cout<<a.get()<<endl;
cout<<b.get()<<endl;
a=b;
cout<<a.get()<<endl;
cout<<b.get()<<endl;
return 0;}
```

Overloading the [] Subscript Operator

In C++, the [] is considered a binary operator for the purpose of overloading. The [] can be overloaded only by a member function. Therefore, the general form of a member operator[]() function is as shown here:

```
type class-name::operator[](int index)
{
// program
}
```

Example:

```
#include <iostream>
using namespace std;
const int SIZE=5;
class arraytype
{
int a[SIZE];
public:
arraytype () { int i; for (i=0;i<=SIZE;i++) a[i]=i;}
int operator[](int i) {return a[i];}
};
int main () {
arraytype ob; int i;
for (i=0;i<=SIZE;i++) cout<<ob[i]<<" ";
return 0;
}
```

It is possible to design the **operator[]()** function in such a way that the [] can be used on both the left and right sides of an assignment statement.

Example:

```
#include <iostream>
using namespace std;
const int SIZE=5;
class arraytype
{
int a[SIZE];
public:
arraytype () { int i; for (i=0;i<=SIZE;i++) a[i]=i;}
int &operator[](int i) {return a[i];}
};
```

```
int main () {  
arraytype ob; int i;  
for (i=0;i<=SIZE;i++) cout<<ob[i]<<" "; cout<<endl;  
for (i=0;i<=SIZE;i++)  
ob[i]=ob[i]+10;// [] ='in solunda  
for (i=0;i<=SIZE;i++) cout<<ob[i]<<" "; cout<<endl;  
return 0;  
}
```

This program displays the following output:

0 1 2 3 4
10 11 12 13 14

Chapter-7 INHERITANCE

Base Class Access Control

When one class inherits another, it uses this general form:

class derived-class-name: access-type base-class-name

```
{  
//Program  
}
```

Here access is one of three keywords: **public**, **private** or **protected**.

- The access specifier determines how elements of the base class are inherited by the derived class.
- When the access specifier for the inherited base class is **public**, all public members of the base become public members of the derived class.
- If the access specifier is **private**, all public members of the base class become private members of the derived class.
- In either case, any private members of the base remain private to it and are inaccessible by the derived class.

Example:

```
#include <iostream>
using namespace std;
class base { private : int x;
public:
void setx (int n) {x=n;}
void showx () {cout<<x<<endl;}
};
class derived :public base { int y;
public:
void sety (int n) {y=n;}
void showy () {cout<<y<<endl;};
int main()
{derived ob;
ob.setx(10);// access member of base class
ob.sety(20);// access member of derived class
ob.showx();// access member of base class
ob.showy();// access member of derived class
return 0;}
```

Example: (Derived class cannot access to the base's private members.)

```
#include <iostream>
using namespace std;
class base { private : int x;
public:
void setx (int n) {x=n;}
void showx () {cout<<x<<endl;}
};
class derived :public base { int y;
public:
void sety (int n) {y=n;}
void show_sum () {cout<<x+y<<endl;} //Error. Cannot
//access private member of base class.
void showy () {cout<<y<<endl;}
};
```

Example: Derived inherits base as private.

```
#include <iostream>
using namespace std;
class base { private : int x;
public:
void setx (int n) {x=n;}
void showx () {cout<<x<<endl;};
class derived :private base { int y;
public:
void sety (int n) {y=n;}
void showy () {cout<<y<<endl;}
};
int main() {derived ob;
ob.setx(10);// Error. Private to derived class
ob.sety(20);//access member of derived class
ob.showx();// Error. Private to derived class
ob.showy();// access member of derived class
return 0;}
```

Example: (Here is a fixed version of the preceding program.)

```
#include <iostream>
using namespace std;
class base { private : int x;
public:
void setx (int n) {x=n;}
void showx () {cout<<x<<endl;}    };
class derived :private base { int y;
public:
void setxy (int n, int m) {setx(n); y=m;}
void showxy () { showx();
cout<<y<<endl; }
int main()  {  derived ob;
ob.setxy(10,20);
ob.showxy();
return 0;
}
```

Using Protected Members

- The protected access specifier is equivalent to the private specifier with the sole exception that protected members of a base class are accessible to members of any class derived from that base.
- Outside the base or derived classes, protected members are not accessible.

The full general form of a class declaration is shown here:

```
class class-name {  
private:  
//private members  
protected:  
//protected members  
public:  
//public members  
};
```

- When a protected member of a base class is inherited as public by the derived class, it becomes a protected member of the derived class.
- If the base is inherited as private, a protected member of the base becomes a private member of the derived class.
- A base class can also be inherited as protected by a derived class. When this is the case, public and protected members of the base class become protected members of the derived class. (Of course, private members of the base class remain private to it and not accessible by the derived class.)

Example:

```
#include <iostream>
using namespace std;
class samp {
private: int a;
protected: int b;
public: int c;
samp (int n, int m) {a=n; b=m;}
int get_a() {return a;}
int get_b() {return b;}
};
int main() {samp ob(10,20);
ob.b=99; //Error. b is protected.
ob.c=30; //c is public.
cout<<ob.get_a<<" "<<ob.get_b<<" "<<ob.c<<endl;
return 0;}
```

Example:

```
#include <iostream>
using namespace std;
class base { protected : int a,b;
public: void setab (int n, int m) {a=n; b=m;}
void showab () {cout<<a<<" "<<b<<endl;}
};
class derived :public base { private : int c;
public:
void setc (int n) { c=n;}
void showabc () {cout<<a<<" "<<b<<" "<<c<<endl;}
};
int main () {derived ob;base ob_2;
ob.setab(1,2); ob.setc(3); ob_2.setab(10,20);
ob.showabc(); ob_2.showab();
return 0;}
```

Example:

```
#include <iostream>
using namespace std;
class base { protected : int a,b;
public: void setab (int n, int m) {a=n; b=m;}
};
class derived :protected base { private : int c;
public:
void setc (int n) { c=n;}
void showabc () {cout<<a<<" "<<b<<" "<<c<<endl;}
};
int main () {derived ob;ob.setc(3);
ob.setab(1,2); //Error setab() is a protected member of
//base and is not accessible here.
ob.showabc();
return 0;}
```

Constructors, Destructors and Inheritance

- It is possible for the base class, derived class, or both to have constructor and/or destructor functions.
- The base class constructor is executed before the constructor in the derived class.
- The reverse is true for destructor function: the derived class's destructor is executed before the base class's destructor.

Example:

```
#include <iostream>
using namespace std;
class base {
public:
base () {cout<<"Constructing base class"<<endl;}
~base () {cout<<" Destructing base class"<<endl;}
};
class derived :public base {
public:
derived(){cout<<" Constructing derived class"<<endl;}
~derived(){cout<<" Destructing derived class"<<endl;}
};
int main () {
derived ob;
return 0;}
```

This Program displays the following output

Constructing base class

Constructing derived class

Destructing derived class

Destructing base class

Example: (passing argument to the derived class's constructor)

```
#include <iostream>
using namespace std;
class base {
public:
base () {cout<<"Constructing base class "<<endl;}
~base () {cout<<"Destructuring base class"<<endl;}
};
class derived :public base { private int j;
public:
derived(int n){j=n;
cout<<" Constructing derived class "<<endl;}
~derived(){cout<<" Destructuring derived class"<<endl;}
void showj() {cout<<j<<endl;}
};
int main () {derived ob(10);
ob.showj();
return 0;}
```

Example: (passing arguments to the both derived class's constructor and base class's consturctor)

```
#include <iostream>
using namespace std;
class base {private : int i;
public:
base (int n) {cout<<" Constructing base class "<<endl; i=n;}
~base () {cout<<" Destructing base class"<<endl;}
void showi() {cout<<i<<endl;}
};
class derived :public base { private : int j;
public:
derived(int n) :base(n) {j=n;
cout<<" Constructing derived class "<<endl;}
~derived(){cout<<" Destructing derived class"<<endl;}
void showj() {cout<<j<<endl;} };
int main () {derived ob(10);
ob.showi(); ob.showj();
return 0;}
```

Example: (In most cases, the constructor functions for the base and derived classes will not use the same argument.)

```
#include <iostream>
using namespace std;
class base {private : int i;
public:
base (int n) {cout<<"Constructing base class"<<endl; i=n;}
~base () {cout<<"Destructing base class"<<endl;}
void showi() {cout<<i<<endl;}
};
class derived :public base { private : int j;
public:
derived(int n,int m) :base(m) {j=n;
cout<<"Constructing derived class"<<endl;}
~derived(){cout<<"Destructing derived class"<<endl;}
void showj() {cout<<j<<endl;} };
int main () {derived ob(10,20);
ob.showi(); ob.showj();
return 0;}
```

Multiple Inheritance

There are two ways that a derived class can inherit more than one base class.

First, a derived class can be used as a base class for another derived class, creating multilevel class hierarchy.

Second, a derived class can directly inherit more than one base class.

When a derived class directly inherits multiple base classes, it uses this expended declaration:

```
class derived-class-name:access base1, access base2,...,
access baseN
{
    Body of class
}
```

When multiple base classes inherited, constructors are executed in the order, left to right, that the base classes are specified. Destructors are executed in the opposite order.

When a class inherits multiple base classes that have constructors that require arguments, the derived class's constructor will be in the following form:

```
Derived-constructor(arg-list) : base1(arg-list) , base2(arg-list) ,..., baseN(arg-list)
{
//body of derived class constructor
}
```

Example:

```
#include <iostream>
using namespace std;
class B1 {private : int a;
public:
B1(int x) {a=x;}
int geta() {return a;}
};
class D1 :public B1 { private : int b;
public:
D1(int x,int y) :B1(y) {b=x;};
int getb(){return b;}
};
class D2 :public D1 { private : int c;
public:
D2(int x,int y, int z) :D1(y,z) {c=x;};
void show(){cout<<geta()<<" "<<getb()<<" "<<c<<endl;}
};
```

```
int main ()
{
D2 ob(1,2,3);
ob.show();
cout<<ob.getA()<<" "<<ob.getB()<<endl;
return 0;
}
```

Example: Derived class directly inherits two base classes

```
#include <iostream>
using namespace std;
class B1 {private : int a;
public:
B1(int x) {a=x;}
int geta() {return a;}    };
class B2 { private int b;
public:
B2(int x) {b=x;};
int getb(){return b;}    };
class D :public B1,public B2 { private : int c;
public:
D(int x,int y, int z) :B1(z), B2(y) {c=x;};
void show(){cout<<geta()<<" "<<getb()<<" "<<c<<endl;};
};
```

```
int main ()
{
D ob(1,2,3);
ob.show();
cout<<ob.getA()<<" "<<ob.getB()<<endl;
return 0;
}
```

Example: The following program illustrates the order in which constructor and destructor functions are called when a derived class directly inherits multiple base classes

```
#include <iostream>
using namespace std;
class B1 {public:
B1 () {cout<<"Constructing B1"<<endl;}
~B1() {cout<<"Destructing B1"<<endl;}
};
class B2 {public:
B2 () {cout<<" Constructing B2"<<endl;}
~B2() {cout<<" Destructing B2"<<endl;}
};
class D :public B1,public B2 {
public:
D () {cout<<"Constructing D"<<endl;}
~D() {cout<<"Destructing D"<<endl;}
};
```

```
int main () {  
D ob;  
return 0;}
```

The program displays the following:

Constructing B1
Constructing B2
Constructing D
Destructing D
Destructing B2
Destructing B1

Virtual Base Classes

A potential problem exists when multiple base classes are directly inherited by a derived class.

Example: Assume

Derived1 class is derived from the **Base** class.

Derived2 class is also derived from the **Base** class.

Derived3 directly inherits *Derived1* and *Derived2*.

- However, this implies that **Base** is actually inherited twice by *Derived3*.
- This causes ambiguity when a member of **Base** is used by *Derived3*.
- To solve this, C++ includes a mechanism by which only one copy of **Base** will be included in *Derived3*. This feature is called *virtual base class*.

Prototype:

```
class derived-class-name: virtual access-specifier base-class-name  
{//Body of the derived class}
```

Example:

```
#include <iostream>
using namespace std;
class A {public: int i;};
class B:virtual public A {public: int j;};
class C:virtual public A {public: int k;};
class D: public B,public C {
public:
int product() {return i*j*k;}
};
int main(){
D ob;
ob.i=10;
ob.j=3;
ob.k=5;
cout<<"Product is"<<ob.product()<<endl;
return 0;
}
```

Bölüm 8 C++'ın I/O Sistemine Giriş

C++' I/O sistemi típkí C'nin I/O sistemi gibi stream (akım) üzerinde hareket eder. Akım bilgi üreten veya bilgiyi alan sanal bir aygıttır. C programı çalıştığında üç akım kendiliğinden açılır (***stdin, stdout ve stderr***). C++'ta ise ***cin, cout cerr ve clog***(ön bellekli cerr) akımları açılır.

C++' I/O sistemi sınıfları

Şablon sınıfı

basic_streambuf

basic_ios

basic_istream

basic_oiostream

basic_iostream

basic_fstream

basic_ifstream

basic_ofstream

8 bitlik karakter tabanlı sınıf

streambuf

ios

istream

oiostream

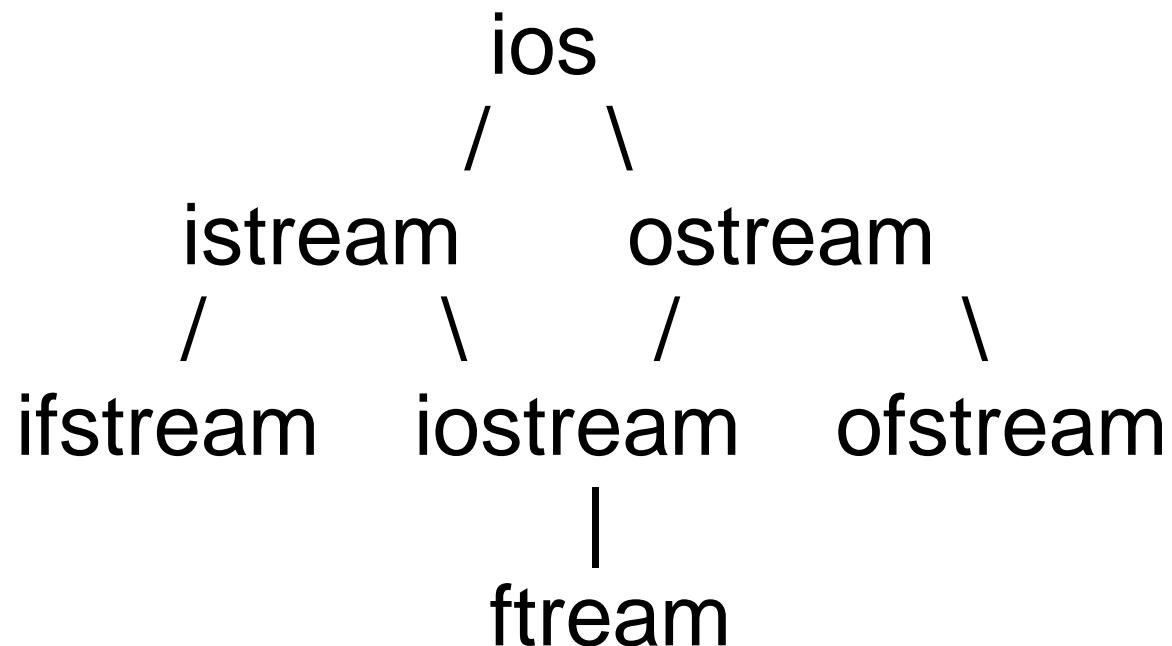
iostream

fstream

ifstream

ofstream

Sınıf türetilme hiyerarşisi



BİÇİMLENDİRİLMİŞ I/O

ios sınıfı içerisinde aşağıdaki değerlerin tanımlandığı fmtflags (enumeration tipinde) bir bitmask deklare edilir.

skipws	basefield	adjustifed	floatfield
unitbuf	hex	left	scientific
boolalpha	oct	right	fixed
showbase	dec	internal	
uppercase			
showpos (pozitif-negatif)			
showpoint (nokta ve sıfırları göster)			
unitbuf (flush output)			

Format bayrağına değer vermek için setf()
fonksiyonu kullanılır. unsetf() setf() fonksiyonunun
tümleyenidir. setf() fonksiyonunun tersini yapar. En
sık karşılaşılan kullanım şekli:

fmtflags setf(fmtflags flags)

Örnek:

```
stream.setf(ios::showpos);  
cout.setf(ios::showbase | ios::hex);
```

unsetf()'in en sık kullanılan prototipi:
void unsetf(fmtflags flags);

Örnek:

```
#include <iostream>
using namespace std;
int main () {cout<<1213.23<<" merhaba "<<<<100<<endl;
cout<<10<<" "<<-10<<endl;
cout<<100.0<<endl;
cout.unsetf(ios::dec);
cout.setf(ios::hex | ios::scientific);
cout<<1213.23<<" merhaba "<<<<100<<endl;
cout.setf(ios::showpos);
cout<<10<<" "<<-10<<endl;
cout.setf(ios::showpoint | ios::fixed);
cout<<100.0<<endl;
cout.setf(ios::showpoint | ios::fixed,ios::floatfield);
cout<<100.0<<endl;
return 0;}
```

`Width()`, `precision()` ve `fill()`'in kullanımı

`streamszie width (streamszie w);`//fonksiyon bir önceki genişliği dönderir:

`streamszie precision (streamszie p);`

`char fill(char ch);` //eski değeri döndürür.

I/O Manipülatörlerinin Kullanılması

Manipulator

boolalpha

dec

endl

fixed

flush

hex

internal

left

noboolalpha

noshowpoint

nounitbuf

noshowpos

noskipws

Input/Output

Input/Output

Output

Output

Output

Output

Input/Output

Output

Output

Input/Output

Output

Output

Output

Input

Manipulator

nouppercase

oct

resetiosflags

right

scientific

setbase

setfill

setiosflags

setprecision

setw

showbase

showpoint

showpos

skipws

unitbuf

uppercase

ws

Input/Output

Output

Input/Output

Input/Output

Output

Output

Input/Output

Output

Input/Output

Output

Output

Output

Output

Input

Input

Output

Output

Input

Örnek:

```
cout<<oct<<100<<hex<<100;  
cout<<setw(10)<<100;  
cout<<setfill('X')<<setw(10);  
cout<<100<<" selam"<<endl;
```

Kendi Inserter'larımıza Oluşturalım

“<<” operatörünün aşırı yüklenmesi: Genel formatı
ostream &operator <<(ostream &stream, sınıf-ismi ob)
{
//program
return stream;
}

Bu bize aşağıdaki gibi yazma imkanı verir

```
cout<<ob1<<ob2<<ob3;
```

Bunu iki şekilde yapabiliriz. Birincisinde aşırı yüklenmiş << operatörü sınıfın arkadaş (friend) olarak tanımlayarak, ikincisinde ise friend olarak tanımlamadan

Örnek:

```
#include <iostream>
using namespace std;
class coord {private: int x,y;
public:
coord () {x=0;y=0;}
coord (int i,int j) {x=i;y=j;}
friend ostream &operator<<(ostream &stream, coord ob);
};
ostream &operator<<(ostream &stream, coord ob)
{
stream<<ob.x<<", "<<ob.y<<endl;
return stream;
}
int main() {coord a(1,1),b(10,23);
cout<<a<<b;
return 0;} //Program çıktısı
1, 1
10, 23
```

Örnek: Aynı programı aşağıdaki şekilde yazabiliriz

```
#include <iostream>
using namespace std;
class coord {
public:
int x,y;
coord () {x=0;y=0;}
coord (int i,int j) {x=i;y=j;}
};
ostream &operator<<(ostream &stream, coord ob)
{
stream<<ob.x<<", "<<ob.y<<endl;
return stream;
}
int main() {coord a(1,1),b(10,23);
cout<<a<<b;
return 0;}
```

Extractor'ların oluşturulması

Genel formatı:

```
istream &operator>>(istream &stream, class &ob)
```

```
{
```

```
//Programın gövdesi
```

```
}
```

Örnek:

```
#include <iostream>
using namespace std;
class coord {private: int x,y;
public:
coord () {x=0;y=0;}
coord (int i,int j) {x=i;y=j;}
friend ostream &operator<<(ostream &stream, coord ob);
friend istream &operator>>(istream &stream, coord &ob);
};
ostream &operator<<(ostream &stream, coord ob)
{stream<<ob.x<<", "<<ob.y<<endl;
return stream;}
istream &operator>>(istream &stream, coord &ob)
{
cout<<"coordinatları girin : ";
stream>>ob.x>>", "<<ob.y;
return stream;}
```

```
int main() {  
coord a(1,1),b(10,23);  
cout<<a<<b;  
cin>>a;  
cout<<a;  
return 0;  
}
```

Bölüm 9 İleri Düzey C++ I/O'su

Kendi Manipülatörlerimizi Oluşturabiliriz

Genel formatı aşağıdaki gibidir.

```
ostream &manip-ismi(ostream &stream)
```

```
{  
//program  
return akım;  
}
```

```
istream &manip-ismi(istream &stream)
```

```
{  
//program  
return akım;  
}
```

```
#include <iostream>
using namespace std;
ostream &setup(ostream &stream)
{
    stream.width(10);
    stream.precision(4);
    stream.fill('*');
    return stream;
}
ostream &atn(ostream &stream)
{stream<<"Attention: "; return stream;}
int main()
{cout<<setup<<123.123456;
cout<<atn<<"High voltage circuit"<<endl;
return 0;
}
```

Dosya I/O Temelleri

Dosya I/O işlemleri yapmak için programınıza `<fstream>` kütüphanesini eklemeniz gerekmektedir. (`#include <fstream>`). Giriş çıkış akımları aşağıdaki gibi oluşturulabilir.

```
ifstream in; //Giriş  
ofstream out; //Çıkış  
fstream in_out; //Giriş ve çıkış
```

Akımi dosyaya birleştirmenin bir yolu `open()` fonksiyonunu kullanmaktır.

Herbir prototip için open() fonksiyonunun prototipi aşağıdaki gibidir.

```
void ifstream::open(const char *filename, open mode=ios::in);
void ofstream::open(const char *filename, open mode=ios::out |
ios::trunc);
void fstream::open(const char *filename, open mode=ios::in |
ios::out);
```

filename : dosyanın ismini gösterir

mode: dosyanın nasıl açılacağını belirler

Mode değerleri:

ios::app dosyanın sonuna ekle

ios::ate dosyanın sonunun aranmasına neden olur

ios::binary

ios::in

ios::out

ios::trunc aynı isimdeki eski dosyanın silimesine neden

Örnek:

```
ofstream mystream;  
mystream.open("test");  
if(!mystream) {cout<<"cannot open file"<<endl;}
```

Mode parametresini girmeye gerek yok çünkü ofstream otomatik olarak output modunda bir stream oluşturur.

Dosyanın açık olup olmadığını **is_open()** fonksiyonu ilede kontrol edebiliriz.

```
if(!mystream.is_open()) {cout<<"File is not open.\n";}
```

open() fonksiyonunu kullanmadanda dosya açılabilir.

```
ifstream mystream("myfile");//Giriş için dosya açılır.  
mystream.close(); // Dosyayı kapatır.
```

bool eof();// Dosyanın sonuna gelinip gelinmediğini gösterir.

Örnek:

```
#include <iostream>
#include <fstream>
using namespace std;
int main ()
{
ofstream fout("test");
if(!fout) {cout<<"Cannot open output file"<<endl; return 1;}
fout<<"Hello"<<endl;
fout<<100<<" "<<hex<<100<<endl;
fout.close();
ifstream fin("test");
if(!fin) {cout<<"Cannot open input file"<<endl; return 1;}
char str[80]; int i,j;
fin>>str>>i>>j;
cout<<str<<" "<<i<<" "<<j<<endl;
fin.close();
return 0;}
```

Örnek:

```
#include <iostream>
#include <fstream>
using namespace std;
int main (int argc, char *argv[])
{if(argc!=2) {cout<<"Usage: deneme dosya_ismi \n";return 1;}
ofstream out(argv[1]);
if(!out) {cout<<"Cannot open output file"<<endl; return 1;}
char str[80];
cout<<"Diske yazı yazar. '$' durmak için"<<endl;
do {
cout<<": ";
cin>>str;
out<<str<<endl;} while (*str!='$');
out.close();
return 0;}
```

```
fstream mystream;  
mystream.open("test",ios::in | ios::out);
```

Eski kütüphanelerde ios tipini belirtmek gerekiyordu.
Şu andaki kütüphanelerde belirtilmediğinde dosya
ios::in | ios::out olarak açılmaktadır.

Biçimlendirilmemiş İkili I/O

En düşük seviyedeki biçimlendirilmemiş I/O fonksiyonları **get()** ve **put()**'tur. En çok kullanım şekli aşağıdaki gibidir.

```
istream &get(char &ch);  
ostream &put(char ch);
```

Veri bloklarını okumak ve yazmak için **read()** ve **write()** fonksiyonları kullanılabilir. Prototipi:

```
istream &read(char *buf, streamsize num);  
ostream &write(const char *buf, streamsize num);
```

Eğer dosyanın sonuna *num* karakterler okunmadan önce ulaşılırsa **read()** durur ve tampon bellek mümkün olduğunda karakter içerir. **gcount()** üye fonksiyonu ile kaç tane karakterin okunmuş olduğunu bulabiliriz. Prototipi aşağıdaki gibidir.

streamszie gcount();

Örnek:(**get()**)

```
#include <iostream>
#include <fstream>
using namespace std;
int main (int argc, char *argv[])
{char c;
if(argc!=2) {
cout<<"Usage: deneme dosya_ismi \n";return 1;
}
ifstream in(argv[1],ios::in |ios::binary);
if(!in) {
cout<<"Cannot open input file"<<endl;
return 1;}
while(!in){in.get(ch);
cout<<ch;
}
in.close();
return 0;}
```

Örnek: (**put()**)

```
#include <iostream>
#include <fstream>
using namespace std;
int main (int argc, char *argv[])
{char ch;
if(argc!=2) {cout<<"Usage: deneme dosya_ismi \n";return 1;}
ofstream out(argv[1],ios::out |ios::binary);
if(!out) {cout<<"Cannot open output file"\n; return 1;}
cout<<"Durdurmak için $ giriniz"\n;
do {
cout<<": ";
cin.get(ch);
out.put(ch);
}while (ch!='$');
out.close();
return 0;
}
```

Örnek: (write() double ve katar yazdırma)

```
#include <iostream>
#include <fstream>
using namespace std;
int main ()
{
    double num=100.45;
    char str[]="This is a test";
    ofstream out("test",ios::out |ios::binary);
    if(!out)
    {
        cout<<"Cannot open output file"<<endl;
        return 1;
    }
    out.write((char *) &num,sizeof(double));
    out.write(str,strlen(str));
    out.close();
    return 0;
}
```

Örnek: (**read()** ile okuma)

```
#include <iostream>
#include <fstream>
using namespace std;
int main ()
{
double num;
char str[80];
ifstream in("test",ios::in |ios::binary);
if(!out)
{
cout<<"Cannot open input file"<<endl;
return 1;}
in.read((char *) &num,sizeof(double));
in.read(str,14);str[14]='\0'
cout<<num<<" "<<str<<endl;
in.close();
return 0;}
```

Biçimlendirilmemiş Diğer I/O Fonksiyonları

*istream &get(char *buf, streamsize num);
istream &get(char *buf, streamsize num, char delim);
int get()*

num-1'e kadar olan karakterler buf'a aktarılır. (yeni satır bulunana kadar veya dosyanın sonuna gelene kadar veya delim karakterine kadar) delim veya yeni satır karakteri akımın içinde kalı buf'a aktarılmaz.

get() akımdan bir sonraki karakteri dönderir. Dosyanın sonuna gelindiğinde EOF dönderir.

*istream &getline(char *buf, streamsize num);*
*istream &getline(char *buf, streamsize num, char delim);*

Get ile getline hemen hemen aynı işi yapar. Tek fark getline() satır sonu, dosya sonu ve delim karakterlerini alır ve siler. get() ise almaz vede silmez.

int peek() akımdan bir sonraki karakteri döndürür fakat akımdan silmez. Dosya sonunda EOF döndürür.

istream &putback(char c); //akımdan gelen son karakteri akıma dönderir.

ostream &flush(); verinin akıma bağlı olan fiziksel aygıta yazılmasını sağlar.

Örnek: (Boşluk içeren bir katarı okumak için getline() kullanılır.)

```
#include <iostream>
#include <fstream>
using namespace std;
int main ()
{
    char str[80];
    cout<<"Enter your name: ";
    cin.getline(str, 79);
    cout<<str<<endl;
    return 0;
}
```

Örnek: ()

```
#include <iostream>
#include <fstream>
#include <cctype>
using namespace std;
int main ()
{
char ch,str[80],*p;
ofstream out("test",ios::out |ios::binary);
if(!out){cout<<"Cannot open output file"<<endl; return 1;}
out<<123<<"this is a test"<<23;
out"Hello there!"<<99<<"sdf"<<endl;
out.close();
ifstream in("test",ios::in |ios::binary);
if(!in){cout<<"Cannot open input file"<<endl; return 1;}
```

```
do {
p=str;
ch=in.peek();
if(isdigit(ch)) {
while(isdigit(*p=in.get())) p++;
in.putback(*p);
*p='\0';
cout<<"Integer :"<<atoi(str);
}
else if(isalpha(ch)) {
while(isalpha(*p=in.get())) p++;
in.putback(*p);
*p='\0';
cout<<"Katar :"<<atoi(str);
}
else in.get(); cout<<endl;//dikkate alma
}while (!in.eof());
in.close; return 0; }
```

Rastgele Erişim

Rastgele erişimi `seekg()` ve `seekp()` fonksiyonları ile gerçekleştirebiliriz.

*istream &seekg(off_type offset, seekdir origin);
ostream &seekp(off_type offset, seekdir origin);*

Seekdir'in alabileceği değerler:

ios::beg

Baştan itibaren ara

ios::cur

Şu anki pozisyonдан itibaren ara

ios::end

Sondan ara

`seekg()` :ilgili dosyanın get pointer'ını belirtilen originden offset kadar hareket ettirir.

`seekp()` :ilgili dosyanın put pointer'ını belirtilen originden offset kadar hareket ettirir.

Her dosya işaretçisinin yerini aşağıda verilen üye fonksiyonlarının yardım ile bulabiliriz.

```
pos_type tellg();  
pos_type tellp();
```

Örnek: (seekp)

```
#include <iostream>
#include <fstream>
using namespace std;
int main (int argc, char *argv[])
{char ch;
if(argc!=4) {
cout<<"Usage: deneme dosya_ismi byte char \n";return 1;}
fstream out(argv[1],ios::in |ios::out | ios::binary);
if(!out) {
cout<<"Cannot open the file"<<endl;
return 1;
}
out.seekp(atoi(argv[2]),ios::beg);
out.put(*argv[3]);
out.close();
return 0;}
```

Örnek: (seekg)

```
#include <iostream>
#include <fstream>
using namespace std;
int main (int argc, char *argv[])
{char ch;
if(argc!=3) {
cout<<"Usage: deneme dosya_ismi yeri \n";return 1;}
ifstream in(argv[1],ios::in | ios::binary);
if(!in) {
cout<<"Cannot open the file"<<endl;
return 1;}
in.seekg(atoi(argv[2]),ios::beg);
while(!in.eof()) {
in.get(ch);
cout<<ch
in.close();
return 0;}
```

I/O Durumunu Kontrol Etme

Bir I/O akımının durumu **iostate** tipinde bir nesnede tanımlanmıştır.

İsmi:

goodbit

eofbit

failbit

badbit

Anlamı:

Hata yok

Dosya sonuna erişilde

Bir I/O hatası oluştu

Fatal bir I/O hatası oluştu

rdstate() fonksiyonu ile I/O durum bilgisini elde edebiliriz. Prototipi

iostate rdstate();

Veya aşağıdaki ios üyesi fonksiyonları çağırabiliriz.

bool bad(); bool eof(); bool fail(); bool good();

Örnek:

```
#include <iostream>
#include <fstream>
using namespace std;
void checkstatus(ifstream &in);
int main (int argc, char *argv[])
{char ch;
if(argc!=) {
cout<<"Usage: deneme dosya_ismi \n";return 1;}
ifstream in(argv[1],ios::in);
if(!in) {
cout<<"Cannot open the file"<<endl;
return 1;}
char c;
while(in.get(c)) { cout<<c; checkstatus(in); }
checkstatus(in);
in.close();
return 0;}
```

```
void checkstatus(ifstream &in)
{
ios::iosstate i;
i=in.rdstate();
if(i&ios::eofbit) cout <<"EOF encountered"<<endl;
else if(i&ios::failbit)) cout <<"Non-Fatal I/O
Error"<<endl;
else (i&ios::badbit)) cout <<"Fatal I/O Error"<<endl;
}
```

Örnek:

```
#include <iostream>
#include <fstream>
using namespace std;
void checkstatus(ifstream &in);
int main (int argc, char *argv[])
{char ch,c;
if(argc!=) {
cout<<"Usage: deneme dosya_ismi \n";return 1;}
ifstream in(argv[1],ios::in);
if(!in) {
cout<<"Cannot open the file"<<endl;
return 1;}
while(in.eof()) { in.get(ch) ;
if(!in.good() & !in.eof())
{cout<<"I/O hata... sonlandırılıyor"<<endl;return 1;} }
cout<<ch;
in.close(); return 0;}
```

Özelleştirilmiş I/O dosyaları

Daha öçe cin ve cout için yazdığımız aşırı yüklenmiş operatörleri (<< ve >>) herhangi bir dosya için de kullanabiliriz.

Örnek:

```
#include <iostream>
using namespace std;
class coord {private: int x,y;
public:
coord () {x=0;y=0;}
coord (int i,int j) {x=i;y=j;}
friend ostream &operator<<(ostream &stream, coord ob);
friend istream &operator>>(istream &stream, coord &ob);
};
ostream &operator<<(ostream &stream, coord ob)
{stream<<ob.x<<", "<<ob.y<<endl;
return stream;}
istream &operator>>(istream &stream, coord &ob)
{
cout<<"coordinatları girin : ";
stream>>ob.x>>", "<<ob.y;
return stream;}
```

```
int main() {
coord o1(1,2),o2(3,4),o3(0,0),o4(0,0);
ofstream out("test");
if(!out) {
cout<<"Cannot open the file"<<endl;
return 1;}
out<<o1<<o2;
out.close();
ifstream in("test");
if(!in) {
cout<<"Cannot open the file"<<endl;
return 1;}
in>>o3>>o4;
in.close();
return 0;
}
```

```
#include <iostream>
using namespace std;
ostream &note(ostream &stream)
{stream<<"Please Note : ";return stream;}
ostream &atn(ostream &stream)
{stream<<"Attention: "; return stream;}
int main()
{
ofstream out("test");
if(!out) {cout<<"Cannot open the file"<<endl; return 1;}
cout<<atn<<"High voltage circuit"<<endl;
cout<<note<<"Turn off all lights"<<endl;
out<<atn<<"High voltage circuit"<<endl;
out<<note<<"Turn off all lights"<<endl;
return 0;
}
```