

Furkan

EKİCİ

2017555017

Object Oriented Programming

Encapsulation: Dışarıdan dataya erisini kapatır. (private, public)

Hoca üzerinde çok durdu.

Polinomifizm: Bir metodun (fonksiyon) veya bir operatörün farklı nesnelerde veya ortamlarda farklı sonuc üretmesidir. Örneğin; C dilinde int iain abs(), long int iain labs(), float iain fabs() kullanılıyor. C++ dilinde ise hepsi için sadexe abs() kullanılıyor. Diğer bir örnek; toplama (+) operatörü iki matrisin toplanması için kullanılabilecek bir operatöre dönüştürülebilir. (Fonksiyon ve operatör arası yüklemesi)

Inheritance: (Kalıtım): Genel class özelliklerini (metodlarını) daha alt classlara aktarmak için kullanılır. Örneğin Person (genel class) diye bir sınıfı olsun ve bu sınıfın kimlik_numarası adında, kimlik numarası alan bir metod olsun. Bir de Student (özel class) adında bir class olsun. Herkeste kimlik numarası olması gerekipinden Person classında bulunan kimlik_numarası() metodunu Student metoduna aktarılır.

Traditional C++

```
#include <iostream.h>  
  
int main () {  
    // code here  
  
    return 0;  
}
```

Modern C++

```
#include <iostream>  
  
using namespace std;  
  
int main () {  
    // code here  
  
    return 0;  
}
```

New Style Headers

```
<iostream>    <vector>
<fstream>    <string>
```

{ in C
 {<math.h>
 {<string.h>

in C++
 {<cmath>
 {<cstring>

C++ Console I/O

cout <<	printf()
cin >>	scanf()

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
```

```
    int i=5;
```

```
    float j=1.2;
```

```
    double x=4.5;
```

```
    cout << i << " " << j << " " << x << endl;
```

→ End line
 ↗ " \n " ile ayrıls.

```
}
```

OUTPUT: 5 1.2 4.5

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
```

```
    int i;
```

```
    cin >> i;
```

```
    cout << "Your number : " << i << endl;
```

```
    return 0;
```

Entered 5

⇒ Output : Your number : 5

! #include "stdafx.h" kütüphanesi sadece Visual Studio'da kullanılır. (Hızlandırmak için)

Classes: A First Look

```
class class-name () {  
private:  
    // private functions and variables  
public:  
    // public functions and variables  
    } object-list;
```

⚠️ Her objenin kendi datası vardır. Fakat kendi fonksiyonu yoktur. Class taki fonksiyonlar objeler tarafından ortak kullanılır.

- * public yazısının üstündeki her şey (private yazılmasa bile) privatedir.
- * Genelde datalar private kısmında olur.
- * Genelde fonksiyonlar public kısmına konur.

Example :

```
#include <iostream>  
using namespace std;  
class myclass {  
private:  
    int a;  
public:  
    void set_a(int num);  
    int get_a();  
}; // Nesne burada da oluşturulabilir di.  
cout << obj.get_a() << endl;  
return 0;
```

OUTPUT = 15

Introducing Function Overloading:

Aynı fonksiyon ismini farklı sayıda veya tipteki verileri kullanmak için fonksiyon adını yüklenir.

Example:

```
#include <iostream>
using namespace std;
int abs1(int n){
    return n<0 ? -n : n;
}
double abs1(double n){
    return n<0 ? -n : n;
}
int main(){
    int x=5;
    double y=10.3;
    cout<<"int number: "<<abs1(x)<<endl;
    cout<<"double number: "<<abs1(y)<<endl;
    return 0;
}
```

* double olan y değişkeni parametre olarak double alan fonksiyona gider, int olan x değişkeni parametre olarak int alan fonksiyona gider.

Example:

```
#include <iostream>
using namespace std;
void date (char *date){ // Burada ilk fonksiyon string,
    cout<<"Date :"<<date<<endl; // ikinci fonksiyon üç adet int
}
void date (int month,int day,int year){ // ifade olur.
    cout<<"Date :"<<month<<"/"<<day<<"/"<<year<<endl;
}
int main(){
    date (" 8/23/2010 ");
    date (8,23,2010);
    return 0;
}
```

Example:

```
#include <iostream>
using namespace std;
f1 (int a) {
    cout << a << endl;
}
f1 (int a, int b) {
    cout << a << " " << b << endl;
}
```

```
int main() {
    f1(10);
    f1(3,7);
    return 0;
```

Introducing Classes

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

class myclass {
private:
    int a;
public:
    myclass () { //Constructor
        cout << "In constructor" << endl;
    }
    ~myclass () { //Destructor
        cout << "In destructor" << endl;
    }
    void set_a (int num) {
        a = num;
    }
    void get_a () {
        return a;
    }
}
```

```
int main () {  
    myclass obj;  
    obj.set_a(15);  
    cout << obj.get_a() << endl;  
    return 0;
```

OUTPUT :

In constructor.

In destructor.

Constructor: Nesnenin datalarına başlangıç değeri verir.

Parametre alabilir. Nesne oluşturduğunda çalışır. Birde fazla kez kullanılabilir. (Fonksiyon asıri yıkama).

Destructor: Genellikle, class dinamik bellek kullanıysa ve bellekte işi bittiyse alınan bellek blokmelerini iade etmeli için kullanılır. Nesnenin kullanıldığı bloktan çıkışker çalışır. Parametre almaz. Bir defa kullanılır.

! İki tane nesne oluşturduğumuzu varsayıyalım (myclass n1, n2;) Öncelikle n1'in constructor'u daha sonra n2'nin constructor'u çalışır. Bloktan çıkışkense önce n2'nin destructor'u daha sonra n1'in destructor'u çalışır (Önce yapılan sonra yıkılır).

Example:

```
int main () {  
    myclass obj1, obj2; }  
    myclass obj3, obj4;  
    return 0;
```

3.

obj1 yapılır, obj2 yapılır, obj2 yokılır, obj1 yokılır,
obj3 yapılır, obj4 yapılır, obj4 yokılır, obj3 yokılır.

Example:

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;
#define SIZE 255;
class Strtype {
private :
    char *p;
    int len;
public :
    Strtype();
    ~Strtype();
    void set(char *ptr);
    void show();
};

Strtype :: Strtype() {
    p = (char *) malloc(SIZE);
    if (!p) {
        cout << "Allocation Error!" << endl;
        exit(1);
    }
    *p = '\0';
    len = 0;
}

void Strtype :: ~Strtype() {
    cout << "Freeing up!" << endl;
    free(p);
}

void Strtype :: set(char *ptr) {
    if (strlen(ptr) >= SIZE) {
        cout << "String too big" << endl;
        exit(1);
    }
    strcpy(p, ptr);
    len = strlen(p);
}

void Strtype :: show() {
    cout << p << " length is " << len << endl;
}

int main() {
    Strtype s1, s2;
    s1.set("This is a test");
    s2.set("I like C++");
    s1.show(); s2.show();
    return 0;
}
```

OUTPUT : This is a test
I like C++
Freeing up. (For S2)
Freeing up. (For S1)

Constructors That Take Parameters:

★ Yapıcı fonksiyon parametre alabilir. Nesne oluşturulurken parantez içinde bu parametreler verilmelidir.

Example:

```
#include <iostream>
using namespace std;
class myclass {
private:
    int a;
public:
    myclass (int);
    void show();
};

myclass :: myclass (int x) {
    cout << "Constructor" << endl;
    a = x;
}

void :: show() {
    cout << a << endl;
}

int main() {
    myclass object (5);
    object.show();
    return 0;
}
```

OUTPUT: Constructor
5.

Introducing Inheritance

```
#include <iostream>
using namespace std;
class B {
private:
    int i;
public:
    void set_i(int n) { i = n; }
    int get_i() { return i; }
};
```

```
class D : public B {
private:
    int j;
public:
    void set_j(int n) { j = n; }
    int mul() { return j * get_i(); }
};
```

```
int main()
{
    B ob1; D ob2;
    ob1.set_i(10);
    ob2.set_i(20); ob2.set_j(30);
    cout << ob1.get_i() << ob2.get_i() << ob2.mul() << endl;
    return 0;
}
```

* B'nin public kısmındaki her element D'nin elementi oluyor.
(B üst class, D alt class)
(kus → papagān)

* B obj'in elementları :

```
private : i
public : set_i, get_i
```

* D obj 'nin elementleri :

```
private : j
public : set_j, get_i, set_i, mul.
```

!"i" D classına mirasla gelir
fakat doğrudan ob2
tarafından erişilemez.

Object Pointers

Classların pointer nesneleri olabilir. Pointer nesneler, elemanlara erişmek için (\rightarrow) operatörünü kullanırlar.

```
My class ob1(35);
```

```
MyClass *p = &ob1;
```

```
cout << ob1.get_i() << endl; >> iki satırda aynı işlevededir.  
cout << p->get_i() << endl;
```

Structures

```
struct type-name {
```

```
    // public func, data.  
private:
```

```
    // private func, data
```

```
} object-list;
```

```
#include <iostream>
```

```
#include <cstring> . . .
```

```
using namespace std;
```

```
struct st-type {
```

```
    st-type (double b, char *n) { balance = b; strcpy (name, n); } // cons
```

```
    void show () { cout << balance << name << endl; }
```

```
} ;
```

```
int main () {
```

```
    st-type st (22, "Furkan");
```

```
    st.show ();
```

```
    return 0;
```

```
3
```

* C'deki struct'tan farklı - (C'de tüm elemanlar public'tır. Burada private olanı erişilemez).

* Private roymadıkça tüm elemanlar public'tır. (classların tensi).

Union

Union'ın tüm üyeleri aynı bellek alanını paylaşır örneğin; Union elemanları içinde int (4 byte) double (8 byte) varsa her elemen için 8 byte ayrıılır. Hem fonksiyon hem de data bulundurabilir. Struct gibi default olarak tüm üyeleri public'tır. Class'ı miras alamaz, class'a miras bırakamaz. Constructor ve Destructor bulundurabilirler.

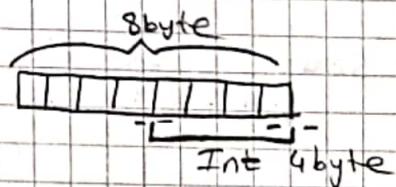
```
#include <iostream>
using namespace std;

union union_x {
    int x; double y;
    void data_show() {
        cout << "The value of x,y is " << endl;
    }
};

int main () {
    union_x u1;
    u1.x=15;
    u1.data_show();
    u1.y=20; u1.data_show();
    return 0;
}
```

Int 4 byte yer kaplar. u1.x=15 de en sağdaki 4 byte'da 15 yazılır. Fakat

u1.y=20'den sonra, 20 sayısı tüm byte'lara dağıllacaktır. En sağdaki byte silinir. (Üstüne yazılır).



In-Line Functions.

C'deki makroya benzer bir mantıkla çalışır. Fonksiyon çağrılarının programda ek olarak getirdiği yükü azaltır. Normal fonksiyona göre hızlı çalışır. Ama boyutu büyükter. Gözde inline fonksiyonu lo kere çağırırsak bellekte lo tane kopya olusur. Inline diyecek compiler'a istek gönderilir. Inline olup olmadığını compiler karar verir. Fonksiyon başına "inline" getirelerek yapılır.

```
#include <iostream>
using namespace std;
int main() {
    int i;
    i = 5;
    even(i);
    i = 6;
    even(i);
    return 0;
}

inline void even(int x) {
    if (!(x % 2)) {
        cout << "Even number" << endl;
    } else {
        cout << "Odd number" << endl;
    }
}
```

Automatic In-Lining: Eğer fonksiyonun boyutu yeteri kadar büyükse bu fonksiyon compiler tarafından otomatik olarak inline haline getirilir. (Class metodları dahil). Örneğin: "Samp" diye bir class olsun.

→ Construction functions da olabilir.

```
samp::samp(int a, int b) {
    i = a;
    j = b;
```

A Closer Look at Classes

* Bir nesneyi diğer bir nesneye atayarak datolarını kopyalayabiliriz

Example:

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

class myclass {
private:
    int a, b;
public:
    void set (int i, int j) { a = i; b = j; }
    void show() { cout << a << b << endl; }
};
```

```
int main() {
    myclass ob1, ob2;
    ob1.set(10, 4);
    ob2 = ob1;
    ob1.show(); // (0, 4)
    ob2.show(); // (10, 4)
    return 0;
}
```

!!! Class'in data kısmında pointer varsa dikkatli ol.

Example: Bu program hatalıdır.

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

class Strtype {
private:
    char *p; int len;
public:
    double y;
    Strtype (char *ptr) {
        len = strlen(ptr);
        p = (char *) malloc(len+1);
        strcpy(p, ptr);
    }
    ~Strtype () {
        cout << "Freeing p.value = " << y << endl;
        free(p);
    }
};
```

```
void show() {
    cout << p << "length is " << len <<
    "the value is " << y << endl;
}

void set (char *ptr, double x) {
    y = x;
    if(strlen(ptr) >= len) {
        len = strlen(ptr)+1; free(p);
        p = (char *) malloc(len);
    }
    len = strlen(ptr)+1; strcpy(p, ptr);
}

// class bitti.

int main() {
    Strtype s1, s2;
    s1.set ("This is a test");
    s2.set ("I like C++");
    s1.show(); s2.show();
    s2 = s1; // Error!
    s1.show(); s2.show();
    return 0;
}
```

* $s2 = s1$ dedikten sonra iki nesnenin datalarındaki pointer da aynı yeri göstermeye başladı. Bu nesneler destruct edilirken önce $s2$ yok edilecek. (Son olusan ilk yıkılır) ve yok olurken p_1 'yi free edecektir. Daha sonra $s1$ 'de yok edilecek ve yok olurken $s1$ 'in p_1 datası da $s2$ 'nin p_1 datasıyla aynı yeri gösterdiğinde aynı yer 2. kez free edilecek ve bu da hatalya yol açacaktır.

Passing Objects to Functions:

Nesnelerin veri olarak fonksiyonlara gönderilmesidir. Örneğin; A nesnesi fonksiyona gönderilirken A'nın kopyası oluşturulup gönderilir. Yeni (kopya) oluşturulan nesnenin constructoru geleneksel constructor'u gölgelir.

* Pointer nesne gönderirsek, nesnenin kendisi pidan ve yapılan değişikliklerden etkilendir. (Normal pointer gibi).

* pointer ile çağırma ($\text{int } *a$) \rightarrow çağırırken $\Rightarrow \text{fonk}(&a)$ (\rightarrow kullan)

referans ile çağırma ($\text{int } \&a$) \rightarrow çağırırken $\Rightarrow \text{fonk}(a)$ (\circ kullan)

```
void f Fonk (myclass x) {
```

```
}
```

```
main () {
```

```
    myclass ob1;
```

```
    f Fonk (ob1);
```

```
    return 0;
```

```
}
```

\rightarrow sonra $ob1$ 'in kendisinin dest'i.

El emanlılar
erişmek
icin.

Example: Normal Yapma

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

class Samp {
private: int i;
public:
    Samp(int n){i=n;}
    void set_i(int n){i=n;}
    int get_i(){return i;}
};

void sqrt_it(Samp o){
    o.set_i(o.get_i() * o.get_i());
    cout << "Value in function: " << o.get_i() << endl;
}
```

```
{ int main(){
    Samp a(10);
    sqrt_it(a);
    cout << "Value in main" <<
        a.get_i() << endl;
    return 0;
}
```

OUTPUT:

Value in function: 100

Value in main : 10.

Example: Pointer ile Yapma

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

class Samp {
private: int i;
public:
    Samp(int n){i=n;}
    void set_i(int n){i=n;}
    int get_i(){return i;}
};

void sqrt_it (Samp * o){
    o->set_i(o->get_i() * o->get_i());
    cout << "in function " << o->get_i() << endl;
}
```

```
{ int main(){
    Samp a(10);
    sqrt_it(&a);
    cout << "in main" <<
        a.get_i() << endl;
    return 0;
}
```

OUTPUT:

in function : 100

in main : 100

Example:

```
#include <iostream>
using namespace std;
class Samp {
private: char *s;
public:
Samp() { s = '\0'; }
~Samp() {
    if (s) free(s);
}
void show() {
    cout << s << endl;
}
void set (char *str) {
    s = (char *) malloc(strlen(str)+1);
    if (!s) { cout << "Allocation Error" << endl;
        exit(1); } // hata olusdu
    strcpy(s, str);
}
Samp input() {
    char s[80]; Samp str;
    cout << "Enter a string"; cin >> s;
    str.set(s);
    return str;
}
```

```
int main () {
    Samp obj;
    obj.set("Hello");
    obj.show();
    return 0;
}
```

Her set edildiginde
obj'e girilen degere
yeni baslangic adresi verilir.
Onceki adres free edilmez.
Bu sebepten bellek dolabilir.

Referans veya pointer
kullanisindak silinti olmazdir.

Set fonk. Esyle olmali:
Uzunluga bat, yeni pelen
string siiyorsa oraya
ya2, sifoniyorsa free
et yeni bellek ayir
oraya ya2.

OUTPUT:
Enter a string: Hello
Freeing s
Freeing s
Hello
Freeing s
Null pointer assignment.

An Introduction to Friend Functions

İki amaçla kullanılır:

(-) Operatör ösiri yüklemesi.

2-) Örnegin iki class birbirlerinin private elemanlarına erişmek isteyebilir. Bu iki class'ı friend yaparak bunu sağlayabiliriz.

Example :

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

class myclass {
private: int n, d;
public:
    myclass (int i, int j){n=i; d=j;}
    friend int isfactor(myclass ob1);
};

int isfactor(myclass ob1){
    if (ob1.n % ob1.d) return 1;
    else return 0;
}

int main()
{
    myclass ob1(10,2), ob2(10,3);
    if (isfactor(ob1)) {
        cout << "2 is a factor of 10" << endl;
    }
    else {
        cout << "2 is not a factor of 10" << endl;
    }
    return 0;
}
```

* Friend fonksiyonları inherit edilemez.

* Bir friend class birden fazla class'a friend olabilir.

* Bir class'in fonksiyonu, başka bir class'a fonksiyon ile arkadaş olabilir.

Chapter 4 - Arrays, Pointers and References -

Example:

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

class Samp {
private:
    int a;
public:
    Samp(int n) { a = n; } // cons
    int get_a() { return a; }
};

int main() {
    Samp objects[4] = { Samp(-1), -2, Samp(-3),
                        Samp(-4) };
    int i;
    for( i=0; i<4; i++ ) {
        cout << objects[i].get_a() << endl;
    }
    return 0;
}
```

* Eğer class'in constructoru varsa elementleri bu şekilde initialize edebiliriz.

* Se setilde de olabilir.

Samp objects[4] = { -1, -2, -3, -4 };

Samp(-4) }

* İki boyutlu ise:

Samp objects[4][2] = { {1, 2, 3}, {4, 5, 6}, {7, 8} } yazarsak.
//consu 1 elemen alıysa objects[0] objects[1]

veya,

Samp objects[4][2] = { Samp(1, 2), Samp(3, 4), Samp(5, 6) }
//consu birden fazla elemen alıysa.

* constructor Samp(int i, int j) { a = i; b = j; }

! Cons ve dest'e dikkat et kesin bir son var!!

Using Pointers to Objects

Example:

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

class Samp {
private: int a,b;
public:
    Samp(int n, int m) {a=n; b=m;}
    int get_a() {return a;}
    int get_b() {return b;}
};
```

```
int main() {
    Samp obj[4] = {Samp(1,2), Samp(3,4),
                    Samp(5,6), Samp(7,8)};
    int i; Samp *p;
    p = obj;
    for(i=0; i<4; i++) {
        cout << p->get_a() << " ";
        cout << p->get_b() << endl;
    }
    p++; // To pass next object.
}
return 0;
```

This Pointer

Objelerin datalarının başlangıç adresini tutan pointer.

```
#include <iostream>
#include <cstring>
using namespace std;
class inventory {
private: char item[20]; double cost;
public:
    inventory(char *i, double c) {
        strcpy(this->item, i);
        this->cost = c;
    }
    void show() {
        cout << this->item << endl;
        cout << cost << endl;
    }
};
```

```
int main() {
    inventory ob("wrench", 4.95);
    ob.show();
    return 0;
}
```

! This yazmasakta compiler onu this formuna çeviriyor.

! This pointer gönderilen nesnenin adresini tutar.

ob2 (....) = ob ((....)) olursa we return this diye
bilsey versa ob1, ob2 ye aktarılır.

Using new and delete

Allocate etmäge yarar.

`malloc() = new , free() = delete`

`p-var = new type;`

`delete p-var;`

Example:

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

int main () {
    int *p;
    p = new int; // Allocate room for an integer.
    if (!p) { cout << "Allocation error" << endl; return 1; }
    *p=1000;
    cout << "Value of *p is " << *p << endl;
    delete p; // release memory.
    return 0;
}
```

Example:

```
#include <iostream>
using namespace std;
class Samp {
private : int a,b;
public:
void set_ab(int i, int j) {
    a=i; b=j;
}
int product () {return a*b;}
};
```

```
} } int main () {
Samp *p;
p = new Samp;
if (!p) { cout << "Allocate Error" << endl;
}
p->set (a,b);
cout << "Product is " << p->product () << endl;
return 0;
}
```

* p-var = new type (initial value), sadece bir nesne olusturulacağsa constructor için type'in sonuna parametre verilebilir.

Şamp *p = new Samp(10, k1); gibi.

p-var = new type [size] ise burada initial value verildiği zaman oluşturulan tüm nesnelere aynı başlangıç adresleri verilmiştir olur. Genelde bu tercih edilmez. Bu diziyi silmek isterseniz;

delete [] p-var; şeklinde free edebiliriz.

Yani,

p=new int [5]; // 5 tane int tutacak kadar yer ayrılr.

delete [] p; free eder.

* class x {

x() { } // Parametre almayan const.

x(int a, int b) { } // Parametre alan const.

}

int main()

x *p, *q;

p=new x((10,20);

q=new x[50];

Buradaki değer genelde galisma zamanında belli olur (Yani 50 değilde herhangi bir değişken olabilir). Buycuda kaç elemanlı olacağı bilinmeyebilir.

Burdan dolayı parametresiz const galisir.

* Dinamik oluşturulan nesnelerin destructurelu silinince çalışır.

Ama nesne dizisi şeklinde ayrılmışsa (new x[10] gibi). Const'un 10 kere dest'i 10 kere halişir. O. indeksin const'u ilk dest'i son kullanır.

p=new x; → p cons
q=new x; → q cons
delete p; → p dest
delete q; → q dest

{ Normalde ilk olusan son yoktur.

References:

Pointer'la benzer bir yapıya sahiptir.

Pointer ile

```
#include <iostream>
using namespace std;
void func(int *n) {
    *n=100;
}
int main() {
    int i=0;
    func(&i);
    cout << "Here is i:" << endl;
    return 0;
}
```

OUTPUT: Here is i : 100

Referans ile

```
#include <iostream>
using namespace std;
void func(int &n) {
    i=100;
}
int main() {
    int i=0;
    func(i);
    cout << "Here is i:" << endl;
    return 0;
}
```

OUTPUT: Here is i : 100.

Referansta sadece fonksiyonun parametre aldığı kısma "&" konur. Diğer tüm değişkenler normal yazılır. Yukarıdaki örnekte "n" (referans örneği) bir pointer değişildir. Bu yüzden fonksiyon için "n"'in başına bir işaret koymadan değişimi değiştirebildik. (n integer'dır.)

Passing References to Functions :

Normal :

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

class myclass {
private: int who;
public:
myclass (int n) {
    who=n;
    cout << "Cons the obj" << who << endl;
}
~myclass () {
    cout << "Dest the obj" << who << endl;
}
int id () { return who; }
};

void f(myclass o) {
    cout << "Received" << o.id() << endl;
}

int main () {
    myclass x(1);
    f(x);
    return 0;
}
```

OUTPUT :

```
Cons the obj
Received 1
Dest the obj
Dest the obj
```

Referans :

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

class myclass {
private: int who;
public:
myclass (int n) {
    who=n;
    cout << "Cons the obj" << who << endl;
}
~myclass () {
    cout << "Dest the obj" << who << endl;
}
int id () { return who; }
};

void f(myclass &o) {
    cout << "Received" << o.id() << endl;
}

int main () {
    Myclass x(1);
    f(x);
    return 0;
}
```

OUTPUT :

```
Cons the obj
Received
Dest the obj
```

| Normal p̄ndirince kopya olusup iki kee dest
galisir. DNA referans ve pointer ile p̄ndirisek
bir kee galisir.

Returning References

Example:

```
#include <iostream>
using namespace std;
int x; //global
int &f() {
    // int x bu satırda yazılısaydı yerel deyişken olurdu.
    return x; // ve program hata verirdi.
}
int main() {
    f() = 100; // assign 100 to reference
    // returned by f().
    cout << x << endl;
    return 0;
}
```

Independent References and Restrictions

Example:

```
#include <iostream>
using namespace std;
int main()
{
    int x = 5;
    int *p = &x;
    int &ref = x;
    cout << x << ref << *p << endl;
    return 0;
}
```

- Bunun özellikleri baktığınızda:
 - Referansın adresini alamayız.
 - Referans dizilleri kuramayız
 - Başka bir referansı referans alamayız.
 -
 - 2 tane data özellüğü var.

OUTPUT: 5 5 5

Birini deyişirsek
hepsi deyişil. Örneğin:
ref = 10; yazarsak hepsi
10 olur.

~ Chapter 5 - Function Overloading ~

Example: (overloading constructor Functions)

```
#include <iostream>           } int main() {
using namespace std;          myclass obj1(10);
                                myclass obj2;
class myclass {                cout << "object1 = " << obj1.getx() << endl;
private: int x;               cout << "object2 = " << obj2.getx() << endl;
public:                         return 0;
myclass() { x=0; }             }
myclass(int n) { x=n; }
int getx() { return x; }
};
```

OUTPUT: object1 = 10
object2 = 0

* Hangi fonksiyonun çağırılacağı compile edilirken belli oluyor.

* Dinamik dizi oluştururken element sayısı galeme anında belli oldupundan (genelde) parametre almayan const yapısı mantıklıdır.

Example:

```
#include <iostream>           void show() {
#include <cstdio>              cout << month << "/" << day << "/" << year << endl;
using namespace std;            // classin sonu.
class date {                   date::date(char *str) {
    int day, month, year;      sscanf(str, "%d%c%d%c%d", &month, &day, &year);
public:                           }
date(char *str);               int main() {
date(int m, int d, int y) {     date sdate("8/23/2010");
    day=d; month=m; year=y;   date idate(8, 23, 2010);
}                                     sdate.show();
};                                         idate.show();
                                         return 0;
```

8/23/2010
8/23/2010

OUTPUT:

~~A~~sscanf fonksiyonu gönderilen string'in içindeki sayıları özel karakter (burada "/") görene kadar sırayla month, day ve year 'in içine atar.

```
sscanf("str-%d%c%d%c%d",&month,&day,&year);  
("10-03-2037") ✓ ("10/03/1978") ✓  
↑ ↑ ↓  
month day year.  
özel özel  
karakter karakter
```

| Class içindeki fonksiyonlarda dinamik bellek kullanılırsa destructor'un içine delete koy. Mainde dinamik nesne veya nesne dizisi kullanılırsa mainin sonuna delete koy.

```
class myclass {  
};  
  
class yourclass {  
private: myclass *p;  
public:  
    yourclass (int i) { //const  
        p = new myclass [i];  
    }  
    ~yourclass () { //dest.  
        delete [] p;  
    }  
};
```

Yukarıdaki açıklamalar ile ilgili örnek.

Creating and Using a Copy Constructor

Üç durumda kullanılır.

- Bir nesneyi, başka bir nesne kullanarak oluştururken.
- Bir nesne, fonksiyona parametre olarak gönderilirken.
- Bir nesne, bir fonksiyon tarafından döndürülürken. (değer olarak).

Genel Format:

classname (const classname & obj) {

 // body of const

original

*p



copy

*p



* Normalde, fonksiyona gönderilirken ve fonksiyondan değer olarak dönerken oluşan kopya nesnenin constructor'u olusmaz demistik. Copy Cons yazarsak eğer galerr.

* Nesne dinamik kullanıyorsa ve yukarıdaki 3 şarttan birini sağlıyorsa copy const yazmak gereklidir.

HATALI

Example: (This program does not use Copy Constructor)

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

class Strtype {
private: char *p;
public:
    Strtype (char*s); //cons
    ~Strtype () { delete []p; }
    Char *get() { return p; }
}; //class sonu.

Strtype::Strtype (char*s) {
    int len = strlen(s)+1;
    p = new char [len];
    strcpy (p,s);
}

void show(Strtype x) {
    Char *s = x.get();
    cout << s << endl;
}

int main() {
    Strtype a("Hello"), b ("There");
    show(a);
    show(b);
    return 0;
}
```

→ a ve b için 2 kere yer ayrılmaya galisilecektir. (1 tane 2 iki diperde kopyası olacak) Show'dan dönerken kopya yıkılacağından hata verir.

Bu hatayı Copy Cons ile çözebiliriz.

(kopya ve ana obje farklı yerde gösterilecektir).

Example: HATASIZ

(This program uses Copy Constructor)

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

class Strtype {
private: char *p;
public:
    Strtype (char*s);
    Strtype (const Strtype &o);
    ~Strtype () { delete []p; }
    Char *get() { return p; }
}; //class sonu.

Strtype::Strtype (char*s) {
    //Yukardakini ayndisi
}
```

```
Strtype::Strtype (const Strtype &o) {
    int len = strlen(o.p)+1;
    p = new char [len];
    strcpy(p,o.p);
} *
```

// Gerisi aynı.

Kopya ayrı bir yeri gösterdiğinde
isin silindi ortadan kaldı.



Using Default Arguments

Example:

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

void func(int a=0, int b=0) {
    cout << a << " " << b << endl;
}

main() {
    func();
    func(10);
    func(10, 20);
}
```

OUTPUT:

0	0
10	0
10	20

Overloading and Ambiguity (Belirsizlik)

Example:

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

```
float f(float i) { return i/2.0; }
```

```
double f(double i) { return i/3.0; }
```

```
main()
```

```
float x = 10.0; double y = 10.0;
```

```
cout << f(x) << endl;
cout << f(y) << endl;
cout << f(10) << endl;
cout << f(10.5);
```

Hata verir! Hangi f fonksiyonu çağırılacak?
 $10 \rightarrow$ double mu?
 $10 \rightarrow$ float mu? (Belirsiz).

10.5'i double olarak kabul eder ve double f fonksiyonuna girer.

Dolayısıyla hata vermez.

* Fonksiyona iki direkt yazılış (3.4, 10.5) gibi sayılar double olarak kabul edilir.

* $\int f(\int a, \int b)$ > maldeki fonksiyon.
 $\int f(\int a, \int \& b)$ > f(10,8) olursa yine belirsizlik var?
 hangi f çağırılacak?

Finding the Address of an Overloaded Function.

`zap()` bir fonksiyon, `ptr`'de bir pointer olsun.
C'de `ptr = zap` yapabiliyorduk. C++'da fonksiyon adını

yuklemesi olduğunu için bunu direkt yapamıyoruz.

Example:

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

void space (int count) {
    for (j; count; count--) cout << ' ';
    cout << "end" << endl;
```

```
void Space (int count, char ch) {
    for (j; count; count--) cout << ch;
    cout << "end" << endl;
```

```
main() {
    Fonk. döner tipi      pointer ismini
    fp1 = space;          fp2 = Space;
    pointer ismi          → Tutulacak fonksiyonun
    void (*fp1) (int);    aldığı parametreler.
```

```
void (*fp2) (int, char);
```

```
fp1 = space;
```

```
fp2 = Space;
```

```
fp1(22); // 22 tane boşluk yazdırır. Sonunda end yazdırır ve aşağıdaki satırı pener
```

```
fp2(10, 'x'); // 10 tane 'x' yazdırır. Sonunda end yazdırır ve aşağıdaki satırı pener
```

```
}
```

$obj = obj + obj \rightarrow obj \cdot \underline{\text{operator}} + (obj)$

$obj \cdot \underline{\text{operator}}(y)$

Chapter 6 - Introducing Operator Overloading

The Basics of Operator Overloading:

Genel yapısı,

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

★ Aslında bir fonksiyondur.

→ . :: . * ?: → Overload yapılmaz

→ 2 arguman alan operatörlerde (+ gibi) iki operand ponderilmeli.

→ Toplama operatörünü asırı yüklediğimizi varsayıyalım. Bu bir fonksiyon olduğu için body kısmında çarpma işlemi yapıp gari döndürebiliriz. Ama bu mantıksız olacağından genelde tercih edilmez.

→ Operatörleri asırı yükleyince işlem ömrüji değismez.

Overloading Binary Operators:

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

class coord {
private: int x,y;
public:
    Bütte 421142252
    Sınıkki Erişk. Coord(int i=0,int j=0){ gidiş
        x=i; y=j;
    }
    void get_xy(int &i, int &j){ i=x -> J=y;
    }
    ~coord()
    {
        coord operator+(coord ob2);
    }
};

coord coord::operator+(coord ob2)
{
    coord temp;
    temp.x = x+ob2.x // this->x+ob2.x
    temp.y = y+ob2.y // this->y+ob2.y
    return temp;
}

main(){
    coord ob1(10,10), ob2(5,3), ob3();
    int x,y;
    ob3 = ob1+ob2; // ob1.operator+(ob2)(Aynı)
    ob3.get_xy(x,y);
    cout<<"(ob1+ob2)"<<x<<x<<"y:"<<y<<endl;
}

```

OUTPUT X:15 Y:13

* coord :: coord operator+(int i) {

 Coord temp;

 temp.x = x + i;

 temp.y = y + i;

 return temp;

} diyerek iki objenin toplanması gibi obje + sayıda toplanabilir.

* Operatorun aynı yüklenmede parametre olarak referans alınabilir.

coord :: coord operator*(coord & ob2)

 Coord temp;

 temp.x = x * ob2.x;

 temp.y = y * ob2.y;

 return temp;

}

* Burada ob3 = ob1 * ob2 yapılmaz (su anlık). (Düzeltenmek için iki yol var: 1 -> Friend Function, 2 -> dataları public yapmak).

Overloading the Relational and Logical Operators:

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;}
    int operator==(coord ob2);
}; // Class sonu
```

```
int coord::operator==(coord ob2)
{
    return (x==ob2.x) && (y==ob2.y);
```

```
} int main()
{
    coord o1(10,20), o2, o3(10,20);
    if (o1==o2)
        cout << "o1=o2" << endl;
    else
        cout << "o1!=o2" << endl;
    if (o1==o3)
        cout << "o1=o3" << endl;
    else
        cout << "o1!=o3" << endl;
}
```

OUTPUT: o1 != o2

o1 = o3

o2 operator == (ob2)

Overloading a Unary Operator

* Döperlerinde iki nesne ile işlem yapıyorduk "+,-,==" gibi.

Burada sadece qapıran nesne var. "++" gibi "-+" gibi.
negatif alma

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;
}

main()
{
    coord o1(10, 10);
    int x, y;
    o1.get_xy(x, y);
    cout << x << y << endl;
}
```

OUTPUT : 11 11

* $ob2 = ++ob1;$ gibi bir atama işlemi varsa return esittir.
(Yukarıdaki örnekte atama olmadı omdatıda returna gerek yoktur)

* Yukarıdaki gibi asırı yükselsek $++'y$ (veya $--$) prefix kullanmayıza

Prefix ve postfix farklı:

Herhangi bir isim olabilir.

Prefix: return-type classname::operator++()

Postfix: return-type classname::operator++(int ix)

→ Postfix'in mantığı budur: gelen nesnenin ilk döperlerini sakla,
nesnenin döperlerini artır, satlana döperi döndür.

```
coord coord :: operator++(int notused) {
```

temp.x = x; temp.y = y; // ilk döperler saklandı.

x++; y++; // döperler artırıldı.

} return temp;

* " $-"$ hem negatif alma hem de çıkarma kullanılır.

coord :: coord operator-() → Negatif alma ($x=-x; y=-y; return *this;$)

coord :: coord operator-(coord ob2) → Çıkarma.

(temp.x = x - ob2.x; temp.y = y - ob2.y; return temp;)

Using Friend Operator Functions

* Friend yapınca qapıran obje dmaz bu nedenle operator kənət operand əlliysə (binary=2 unary=1) bunları parametrlər olaraq yollanır.

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);
};

// Class sonu
```

```
{ coord operator+(coord ob1, coord ob2) {
    coord temp;
    temp.x = ob1.x + ob2.x;
    temp.y = ob1.y + ob2.y;
    return temp;
}

main()
{
    coord ob1(0,10), ob2(5,3), ob3;
    int x,y;
    ob3 = ob1 + ob2;
    ob3.get_xy(x,y);
    cout << x << y << endl;
}
```

OUTPUT: 15 13

* Overloading Binary Operators baslıqının altında son olaraq int+obj yapmak işin friend func. kullanabiliriz domistik

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+(int i, coord ob1);
    friend coord operator+(coord ob1, int i);
    friend coord operator+(coord ob1, coord ob2);
};

// Class sonu
```

* Artık coord+coord
coord+int və
int+coord yapabiliyiniz.

$$\begin{aligned} 3 + 0.1 \\ 0.1 \rightarrow 5 \\ 0.2 + 0.5 \end{aligned}$$

* Unary operatörleri asırı yüklenken parametre olarak referans almalıdır. Referansla gönderme sebebi içindi de örneği de göstermez. Lütfen fonksiyonlarda olduğunu gibi kopyasını gönderiyoruz.

Example:

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

class coord { private 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 &obj);
}; // Class sonu
```

```
{ coord operator++(coord &obj) {
    obj.x++;
    obj.y++;
    return obj;
}

main() {
    coord o1(10,10); int x,y;
    ++o1; // prefix
    // o1 referansı yolladı
    o1.get_xy(x,y);
    cout << x << y << endl;
}
```

OUTPUT: 11 11

* Yukarıdaki örnekte prefix kullanım vardır.

Prefix: coord operator++(coord &obj);

Postfix: coord operator++(coord &obj, int not used);

! Yukarıdaki örnekle referans ile de döndüribiliyorduk.

A Closer Look at the Assignment Operator:

Copy constructor'da veya normalde dinamik bellek kullanınan nesnenin başka bir nesneye aktarımında sorun oluyordu. Bu sorunu düzeltmek için kullanılır.

Example :

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

class Strtype {
private: char *p; int len;
public:
    Strtype(char *s) { // constructor
        int l;
        l = strlen(s) + 1;
        p = new char[l];
        strcpy(p, s);
        len = l;
    }
    ~Strtype() { delete []p; } // destructor
    char *get() { return *p; }
};

Strtype &operator=(Strtype&ob) {
    delete []p;
    p = new char[ob.len];
    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;
}
```

// OUTPUT : Hello
There
There
There

* Copy Constructor kullanısaydık value olarak alabilirdik.
Kullanmadığımız için referans ile alındı.

Overloading the [] Subscript Operator:

Genel kullanım:

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

Example:

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

* Örnekte, nesne kullanılarak dizinin elementleri alınmıştır.

Example:

Eğer fonksiyon: int &operator[](int i) { return a[i]; }

Eğerinde olsaydı sadece değer almaktır kalmayıp aynı

zamanda mainde deger de atayabiliirdik. (Referans olduğunu için)

```
for (i=0; i<SIZE; i++)  
    obj[i] = obj[i] + 10; // []'in solunda
```

OUTPUT	0	1	2	3	4
10	11	12	13	14	
20	21	22	23	24	
:	:	:	:	:	

Chapter 7 - Inheritance -

Base Class Access Control

Genel yapı:

```
class derived-class-name: access type base-class-name {  
}; //program
```

→ Access type iin 3 keyword vardir = public, private, protected.

★ Access type ne olursa olsun alt class, üst class'in datalarına doğrudan erişemez. Erişmek iin set, get gibi fonksiyonlar tanımlayıp onları kullanmanız lazımdır.

Base class dataların access typeleri	Access type	Erişibilirlik
---	-------------	---------------

public	public protected <u>private</u>	public protected <u>private</u>
protected	public protected <u>private</u>	protected protected <u>private</u>
private	public protected <u>private</u>	private private <u>private</u>

★ Bir temel sınıfın **public** üyelerine o sınıfın içindeinden, programda herhangi bir yerden veya sınıfın türetilmiş sınıflarından erişilebilir.

★ Bir temel sınıfın **private** üyelerine sadece sınıfın içindeinden ve o sınıfın arkadaş sınıflarından erişilebilir.

★ Bir temel sınıfın **protected** üyeleri, sınıfın içinde, sınıfın arkadaş sınıflarının üyelerinde ve temel sınıfın türetilen herhangi bir sınıf ve o sınıfın arkadaş sınıflarının üyelerince erişilebilir.

Example:

```
#include <iostream>
using namespace std;
class base { private: int x;
public:
    void setx (int n) { x=n; }
    void show() { cout << x << endl; }
}; //base class sonu.

class derived : public base { int y;
public:
    void sety(int n) { y=n; }
    void showy() { cout << y << endl; }
}; //derived class sonu.

int main() { derived ob;
    ob.setx(10); //access member of base class
    ob.sety(20); // " " " derived "
    ob.showx(); // " " " base "
    ob.showy(); // " " " derived "
    return 0;
}
```

*class derived : private base derset,

ve mainde derived ob tanımlasın :

ob.setx yazamayız mainin içine. Ama derived classının possibilitànden base classın public elemanlarına erişebiliriz. Neydi?

derived ob1;

base ob2; tanımlayalım mainde, (yine inherit edilmiş olsun)

ob1.setx(10); → yazamazken

ob2.setx(10); → yazabiliyoruz.

Cünkü inherit eritken verdigimiz access type private.

Bu nedenle derived kullanarak base'in public elemanlarına erişilemez. Çünkü artık private oldular. (page 2 sayfası tablo). base classı ise bunlara baþlı olmadığını den ob2 ile public elemanları erişim sağlanabilir.

Example: (HATALI)

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

class base { private: int x; }
public:
    void setx(int n) { x=n; }
    void showx() { cout << x << endl; } //base class sonu

class derived : private base { private: int y; }
public:
    void sety(int n) { y=n; }
    void showy() { cout << y << endl; } //derived class sonu.

int main() { derived ob;
    ob.setx(10); //Error. Private to derived class.
    ob.sety(20); //Access members of derived class
    ob.showx(); //Error. Private to derived class
    ob.showy(); //Access member of derived class
    return 0;
}
```

Example: (DÜZELTİLMİŞ)

Class base → Aynı.

```
class derived : private base { private: int y; }
public:
    void setxy(int n, int m) { setx(n)=n; y=m; }

    void showxy() { cout << y << endl; }

};

class Sonu.

main() { derived ob;
    ob.setxy(10, 20);
    ob.showxy();
    return 0;
}
```

Using Protected Members

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 : public base {private: int c;
public:
    void setc (int n){c=n;}
    void showabc(){cout<<a<<b<<c<<endl; // a ve b ye
};//derived sonu
};

int main ()
{
    derived ob; base ob_2;
    ob.setab(1,2); ob.setc(3); ob_2.setab(10,20);
    ob.showabc(); ob_2.showabc();
    return 0;
}
```

Example :

Base aynı, class derived :protected base olsaydı, derived
için de aynı olsaydı :

main'e ob.setab(10) yazamazdık. Çünkü access type
protected olduğundan dolayı yoldan base fonsiyonlarına
(public olan) erişmeye fırsatıca protected gibi davranış.
Ama, ob_2.setab desek bir sıkıntı olmasın.
Çünkü public.

Constructors, Destructors and Inheritance

Example:

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

class base {
public:
    base() { cout << "Const. base class" << endl; }
    ~base() { cout << "Dest. base class" << endl; }
};

class derived : public base {
public:
    derived() { cout << "Const. derived class" << endl; }
    ~derived() { cout << "Dest. derived class" << endl; }
};

int main() {
    derived obj; // Const. derived class
    return 0; // Dest. derived class
} // Dest. derived class.
```

//OUTPUT: Const. base class
Const. derived class
Dest. derived class
Dest. derived class.

Buradaki mantık: önce en kapsamlı olan class'ın constructoru
göllür. Dest. galisim sırası ise tam tersidir.

* derived(int n) : base(n) { }
base(n)

Bu ifade derived classının constructor'ıdır. buradaki base(n)
int n. deki n değerini basenin constuna yollar. Sosyal
parametresi içinde ise derived classının J elemanı
n değerini alır.

derived(int n, int m) : base(n) { }
base(n)

Böyle de kullanılır.
n değeri basaya, m değeri derived'e gider.

Multiple Inheritance:

Genel yapı:

class derived-class-name : access base1, access base2, ...

* Bir class, istedipki kadar class miras alabilir.

Const kuralıma sırası → B1, B2, ..., D.

Example :

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

```
class B1 { public:
```

```
    B1() { cout << "Constr B1" << endl; }
```

```
    ~B1() { cout << "Destr B1" << endl; }
```

```
};
```

```
class B2 { public:
```

```
    B2() { cout << "Constr B2" << endl; }
```

```
    ~B2() { cout << "Destr B2" << endl; }
```

```
};
```

```
class D : public B1, public B2 { public:
```

```
    D() { cout << "Constr D" << endl; }
```

```
    ~D() { cout << "Destr D" << endl; }
```

```
};
```

```
int main () {
```

```
    D obj;
```

```
    return 0;
```

OUTPUT :

Constr	B1
Constr	B2
Constr	D
Destr	D
Destr	B2
Destr	B1

Example :

```
class B {
```

}, olsaydı ve main içinde D2 obj; olsaydı.

```
class D1 : public B {
```

} (D2 dolaylı yoldan B1'i de alır).

```
class D2 : public D1 {
```

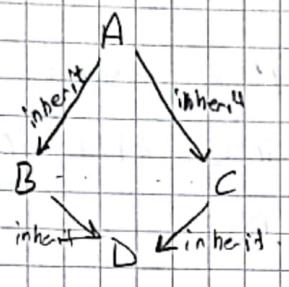
OUTPUT :

Constr	B1
Constr	D1
Constr	D2
Destr	D2
Destr	D1
Destr	B1

* Dinamik bellek kullanılmışsa ve
delete versa, delete sırasına
göre silinir.

~ Virtual Base Classes ~

Eğer;



→ B, A'nın datalarını miras alıyor.
→ C'de A'nın datalarını miras alıyor.
→ D hem B, hemde C'den dolayı
 A'yı dolaylı yoldan iki kez miras
 alıyor.

→ Bunu önlemek için access type öncesi (B ve C tanımlanırken)
A'nın öncesi "virtual" ifadesi koymalıdır.

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 obj;
    obj.i = 10;
    obj.j = 5;
    obj.k = 2;
```

```
    cout << "product is " << obj.product () << endl;
```

```
    return 0;
}
```

! - CHAPTER 8 ve 9 işlenmedi - !

Chapter 10 - Virtual Functions

→ Polymorfizm (ösin yükleme) yapmak için 2 yol vardır:

- 1-) Çalışma zamanında karar verilen (Virtual Functions bunun için kullanılır)
- 2-) Compile zamanında karar verilen.

Base class'ın bir pointer'ı olsun. Bu pointer bu baseyi miras alan derived class'lar tarafından da kullanılabilir.

```
base *p;  
base base_obj;  
derived derived_obj;  
p = &base_obj;  
p = &derived_obj;
```

Example: (Bu örneğin çıktısına bak)

```
#include <iostream>  
using namespace std;  
  
class base{private:int x;}  
public:  
void set_x(int i){x=i;}  
int get_x(){return x;}  
  
class derived:public base{private:int y;}  
public:  
void set_y(int j){y=j;}  
int get_y(){return y;}  
  
int main(){  
base *p, b_obj;  
derived d_obj;  
p = &b_obj;  
p->set_x(10);  
cout << "Base x" << p->get_x() << endl;  
  
p = &d_obj;  
p->set_x(99);  
cout << "Derived x" << p->get_x() << endl;  
cout << "Derived y" << p->get_y() << endl;  
return 0;  
}
```

OUTPUT: 10 99

* Pointer ile önce basenin x'ine erişip değıstırılıyor. Sonra derived'in x'ine erişip değıstırıyoruz.

* base *p yazarsak (yükarıdaki gibi) ve p = derived_obj yazarsak.
p kullanırsak derived baseden inherit edilen datalara erişebiliyoruz yani;
p->set_y(100) → hatalıdır. Eğer derived *p ve p = derived_obj
olsaydı p->set_y(100) doğru olurdu.

Introduction to Virtual Functions

- * Fonksiyon öncine "virtual" keyworduyla yapıllır.
- * Base class'in içinde tanımlanır. Derived classlar tarafından yeniden tanımlanır.

Example:

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

class base {
public: int i; base(int x) {i=x;}
virtual void func() {
    cout << "base func:" << i << endl;
}
}
```

```
class derived1 : public base {
public: derived1(int x) : base(x) {}
void func() {
    cout << "Derived1 func:" << i << endl;
}
}
```

OUTPUT: 10, 100, 20

```
class derived2 : public base {
public: derived2(int x) : base(x) {}
int main() {
    base *p, b=ob(10);
    derived1 d1=ob(10);
    derived2 d2=ob(20);

    p=&b;
    p->func();

    p=&d1;
    p->func();

    p=&d2;
    p->func();

    return 0;
}
```

! D2'de fonksiyon tanımlı değil. Bu yüzden basedeki fonksiyonu kullanır. Eğer derived 2'de;
void func() {
 cout << "Derived2 func :" << i << endl;
}

yazılıdı OUTPUT: 10, 100, 20 olurdu.

- * Fonksiyon deiri yıklanmadede ya dańız tipi ya da alındığı parametrelerin tipi veya sayısı değişik oluyordu. Burada her şey aynı.
- * Virtual Function'lar bir class'a üye olmaya (Base class)

Example:

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

class area {double dim1, dim2;
public:
    void setdim(double d1, double d2) {
        dim1 = d1;
        dim2 = d2;
    }
    void getdim(double& d1, double& d2) {
        d1 = dim1;
        d2 = dim2;
    }
    virtual double getarea() = 0;
    cout << "You must override this function" << endl;
    return 0.0;
};

class rectangle: public area {
public: getarea() {double d1, d2;
    getdim(d1, d2);
    return d1 * d2;
};
}
```

```
class triangle: public area {
public: getarea() {double d1, d2;
    getdim(d1, d2);
    return 0.5 * d1 * d2;
};

int main() {
    area *p;
    rectangle r; r.setdim(10, 5);
    triangle t; t.setdim(4.0, 5.0);
    p = &r;
    cout << "Rectangle A:" << p->getarea() << endl;
    p = &t;
    cout << "Triangle A:" << p->getarea() << endl;
    return 0;
}

OUTPUT: Rectangle A: 50
                    Triangle A: 10
```

More About Virtual Functions

General form: virtual type func-name (parameter-list) = 0.

Above example, base class's implementation of getarea is required.

Annas declaration is optional. In the above example, o class:

virtual double getarea () = 0

can be written. This is called pure (safe) virtual function.

* Pure function's derived classes must be defined as public.

Otherwise, error occurs. If there is no derived class, it is not mandatory.

— DÖNEM SONU —