

Objektbaserad programmering (i C++)

klasser , objekt och operator-överlagring

Nayeb Maleki

Programming paradigm

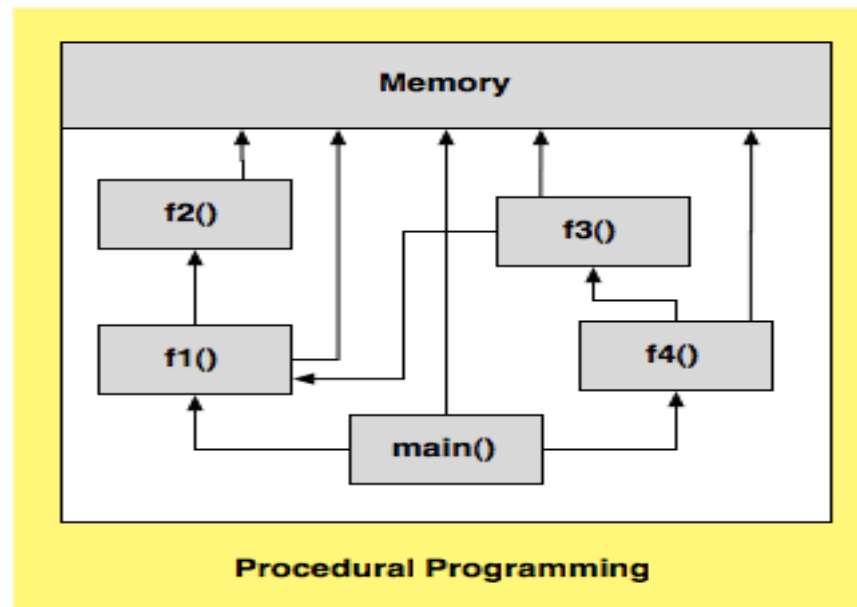
- Definierar design- och implementeringsmetodiken, inklusive språkets byggstenar, interaktionen mellan datastrukturer och operationer mot dessa, hur problem generellt kan analyseras och lösas. Programmeringsparadigm är "teori:n" om ett (flera) programmeringsspråk.
- Ett programmeringsspråk specificerar nyckelord, preprocessor direktiv, program struktur, standard bibliotek, etc. som stöd för en programmeringsparadigm.
- "Normalt" ett givet prog. språk syftar till ett specifikt applikationsområde (domän): web programmering, string manipulering, simulering, etc.
- Och C++?

Programming paradigm

- C++ är inte bunden till en specifik applikationsdomän.
- Språket har stöd för flera olika (ngt. lika!) viktiga programmeringsparadigm:
 - Procedurell programmering (C , Pascal, Fortran, ...)
 - Objekt-Baserad programmering
 - Objekt-Orienterad programmering
 - Generisk (generic) programmering (för att ge ytterligare (högre) nivå av återanvändbarhet som är ngt. begränsad i OOP.)

Procedurell Programmering

- Procedurell programmering grundar sig på separation mellan funktioner och det data dessa manipulerar. Generellt, funktioner är beroende av den fysiska representationen av datatyper dessa manipulerar. Detta djupa beroende är en utav de mest problematiska aspekterna i underhåll och vidare utveckling av mjukvaror.



Procedurell Programmering

- Procedurell programmering är känslig mot design förändringar. Om och när definitionen av en typ ändras (tex. p.g.a ändringar i kundernas villkor, eller för att anpassa mjukvaran till olika plattformar, ...), så måste ”alla” funktioner som refererar till den typen också modifieras därefter. Motsatsen är också giltig självklart, när en funktion ändras, påverkas också dess argument .
- **MEN**
- Procedurella programmeringsspråk producerar dem mest effektiva maskinkoden jämfört med andra högnivåspråk. Faktum är att vissa har detta som ett av sina starkaste motiv mot objektorienterade språk.

Procedurell Programmering

- I procedurell programmering:
 - koncentrationen är kring operationer (procedurer, funktioner). känslig mot design förändringar.
 - Data kommer i andra hand.
- Svårt att relatera till verkligheten
 - Det finns inga funktioner i verkligheten
- Svårt att skapa nya datatyper, detta minskar utvecklingsmöjligheter.

Objekt-Baserad Programmering

- Begränsningarna i Procedurell programmering ledde forskarna till en paradigm som gjorde att man nu kunde på ett bättre och effektivare sätt separera implementationsdetaljer från gränssnitt. Användardefinierade typer packas nu med data och meningsfulla operationer som kallas för **en klass**, som också har stöd för **informationsgömning**, vilket ledde till separering av detaljer från de samling tjänster som klassen ger, dess gränssnitt (Interface).

Objekt-Baserad eller Objekt-Orienterad Programmering?

- Objekt-Baserad programmering är Objekt-Orienterad programmering utan arv och stöd för basklasser. Alltså man kan lungt påstå att Objekt-Baserad programmering är en delmängd av Objekt-Orienterad programmering.

Objekt-Baserad eller Objekt-Orienterad Programmering?

- Att Objekt-Baserad programmering inte har nämnda egenskaper är inte en tillfällighet. Man brukar motivera med att existensen av de egenskaperna ger en komplex design, basklasser kan vidarebefodra buggar till underklasser samt att **Arv** leder till **polymorfism** vilket i sin tur komplicerar mjukvaru-designen ännu mer. Tex. en funktion som tar ett objekt av en basklass som argument, vet också hur ett (publikt) härlett objekt från den basklassen kan hanteras.

Objekt-Baserad eller Objekt-Orienterad Programmering?

- Men Objekt-Baserad programmering har också **svagheter**. Man kan inte på ett direkt sätt fånga sambanden mellan olika objekt som existerar i verkligheten. Tex. likheterna mellan en hårddisk och en diskette. Båda kan lagra filer, innehålla kataloger och underkataloger, etc. Så i implementeringen måste man skapa två skilda objekt utan hänsyn till de gemensamma egenskaperna..

Objekt-Orienterad Programmering

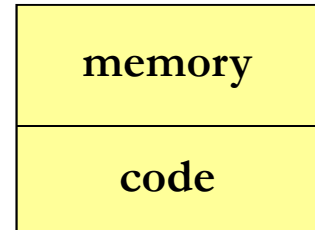
- Den gemensamma nämnaren för alla Objekt-Orienterade programmeringsspråk är:
- **Inkapsling** (Encapsulation/Information hiding/Data abstraction): som menar att implementationsdetaljer av en klass är gömda från den som använder klassen.
- **Arv (Inheritance)**: som ger möjligheten att ett härlett objekt **återanvända** funktionalitet och gränssnitt från bas-klassen. Fördelen är enorma: kortare utvecklingstid, enklare underhåll och vidareutveckling.

Objekt-Orienterad Programmering

- **Polymorfism (Polymorphism):** mångformighet innebär att beroende på de ingående operanderna, en (program)konstruktion som ser ut på ett visst sätt kan innebära olika saker(tex. anrop av olika funktioner). Tex. när man säger: ”stäng” till skilda objekt, det kan betyda: stänga dörren, bankkonto eller ett programfönster. Skilda reaktioner från skilda objekt för samma meddelande.
- C++ har tre olika mekanismer av **statisk**(compile-time) **polymorfism: funktionsöverlagring, operator överlagring, templates(mallar)**

Klasser och Objekt

- En klass är en användardefinierad datatyp.
- En klass är en samling objekt (i verkligheten) med likadana egenskaper och beteenden.
- En klass (i C++ till exempel) är en mall för att skapa objekt.
- En klass är en Abstrakt DataTyp.
- Klasser är datatyper vilkas värde är objekt!
- Objekt-Orienterad programutveckling börjar med att under Analys & Design fasen leta efter (rätta) klasser. Detta är en utav de absolut viktigaste fasen i OOP-utveckling.



Klassdefinition: generell syntax

class <classname>

{

public: //Synligt område

<datatyp> variabelnamn;

<datatyp> variabelnamn;

<returtyp> <funktionsnamn> (parametrar);

<returtyp> <funcktionsnamn> (parametrar);

private: //Gömt område

<datatyp> variabelnamn;

<datatyp> variabelnamn;

<returtyp> <funktionsnamn> (parametrar);

<returtyp> <funktionsnamn> (parametrar);

//protected:

};

Klass-definition och implementation:

Exempel 1: Student.cpp

```
class Student {  
public:
```

```
void setNumber(long); // deklaration av en operation (medlemsfunktion)  
long getNumber() const; // deklaration av en operation (medlemsfunktion)
```

```
private:
```

```
long number_; // attribut(datamedlem)  
}; // semikolon
```

```
// Definition av medlemsfunktioner
```

```
long Student::getNumber() const { // Query operation //getter  
    return (number_);  
}
```

```
void Student::setNumber(long number) { // modifier operation //setter  
    number_ = number;  
}
```

Query och Modifier medlemsfunktioner

- Ökar läsbarheten
- Kompilatorn hjälper dig med felaktig kod:

```
long Student::getNumber() const {  
    number_ += 100;           // kan ej ändra objektet  
    return number_;  
};
```

- Konstanta medlemsfunktioner kan anropa *andra medlemsfunktioner* endast om dessa är konstanter
- Endast konstanta medlemsfunktioner kan anropas för objekt som är deklarerade som *konstanter*.

Klassdefinition: Exempel 2: Stack.h

```
#ifndef _STACK_H_
#define _STACK_H_
#include <iostream>
#include <cstring>
#include <cassert>
using namespace std;
const int STACK_SIZE=10;
class stack {
    private:
        int count; // Number of items in the stack
        int data[STACK_SIZE]; // The items themselves
    public:
        void init( ); // Initialize the stack
        void push(const int item); // Push an item on the stack
        int pop( ); // Pop an item from the stack
};
#endif
```

Exempel 2 forts. : Stack.cpp

```
#include "Stack.h"
```

```
void stack::init() {  
    count = 0;  
}
```

```
void stack::push(const int item) {  
    // assert((count >= 0) && (count < sizeof(data)/sizeof(data[0]) ));  
    data[count] = item;  
    ++count;  
}
```

```
int stack::pop() {  
    --count;  
    //assert((count >= 0) && (count < sizeof(data)/sizeof(data[0])));  
    return (data[count]);  
}
```

Exempel 2 forts.: Stack_main.cpp

```
#include "Stack.h"
```

```
int main( ) {
```

```
    stack a_stack;
```

```
    a_stack.init();
```

```
    a_stack.push(1);
```

```
    a_stack.push(2);
```

```
    a_stack.push(3);
```

```
    cout << "Expect a 3 ->" << a_stack.pop() << endl;
```

```
    cout << "Expect a 2 ->" << a_stack.pop() << endl;
```

```
    cout << "Expect a 1 ->" << a_stack.pop() << endl;
```

```
    return (0);
```

```
} //end-main
```

Automatiskt definierade medlemsfunktioner

Dessa definieras (för varje klassdef.) och anropas automatiskt (för varje objekt) där det krävs, om man inte definierat egna, vilket man bör göra.

```
klassnamn( );          //Default konstruerare (konstruktor)
//The Big Tree Rule
~klassnamn( );         // "Default" destruerare (destruktor)
//The Big Two Rule
klassnamn(const klassnamn& ); //Kopieringskonstruerare
Samt operator= ( Tilldelningsoperatorn )
klassnamn & operator= (const klassnamn &);
```

Automatiskt definierade medlemsfunktioner

(Default) Konstruktör

- Objekt initieras med **konstruktörer**
- **Konstruktörer** kan överlagras
- **Konstruktörer** har inga returvärden
- **Konstruktörer** har samma namn som klassen de tillhör
- En konstruktör **anropas** när:
 - Objektet skapas **statiskt** → **Student S;**
 - Objektet skapas **dynamiskt** → **Student *S=new Student;**
- Konstruktorn **anropas inte** vid värdeanrop eller vid funktionsretur

Parametriserade Konstruktorer

```
//i book.h
class book {
    string author;
    string title;
    int status;
public:
    book (string n, string t, int s); // Parametriserad Konstruktor
    int get_status() {return status;}
    void set_status(int s) {status = s;}
};

//i book.cpp
book::book (string n, string t, int s) {
    author=n;
    title=t;
    status = s;
}

//i main.cpp
book b1 ("Twain", "Tom Sawyer", 1);
book b2 ("Melville", "Moby Dick", 0);
```

Automatiskt definierade medlemsfunktioner

(Default) Destruktor

- Destruktorer används för att frigöra **dynamiskt allokerade minnesutrymmen**.
- En destruktor får **inte** anropas eller returnera ett värde.
- Destruktorn anropas automatiskt när objektet upphör att existera (t.ex. ett lokalt objekt i en funktion).

Kopieringskonstruktor:

klassnamn(const klassnamn& p);

där **p** är en const-referens till the objekt som används för att initiera det nya objektet.

Anropas när:

- Ett objekt initieras genom att tilldela det, ett redan existerande objekt.
Student a=b;
- Den formella parameter initieras genom att använda **värdet** av den aktuella parameter. **Student v; funktion(v);**
- Ett objekt returneras från en funktion. Till exempel:
Student val = fun();

Om ett objekt **inte** har **pekare** till vilken dynamisk minnesarea skall skapas, en grund kopiering (som default utförs) brukar räcka.

Om du behöver kopieringskonstruktorn , behöver du också operatör **=**
Därför att du då behöver en djup kopiering.

Klassdefinition: Exempel 1: omskrivning 1

//Klassdefinition: Student.hxx (Student.h)

```
#ifndef STUDENT_H
#define STUDENT_H
```

//Include:er kommer här!

```
class Student {
```

```
public:
```

```
    Student( ); //deklaration av default konstruerare
```

```
    Student(long number) ; // dekl. av en konstruerare
```

```
    //Student(long number=0):number_(number){} // dekl. & def. av en //konstruerare
                                           //med en imparam. som skall ersätta de två ovan
```

```
    ~Student( ){ }; // deklaration & def. av default destruerare
```

```
    long getNumber( ) const; // Accessor/Query /getter
```

```
    void setNumber(long); // mutator /Modifier /setter
```

```
private:
```

```
    long number_; // attribut(datamedlem)
```

```
}; // semikolon
```

```
#endif
```

Klassimplementations:Exempel 1:omskrivning 1

```
// Implementering av medlemsfunktioner : Student.cxx
#include "student.hxx"
long Student::getNumber() const {          //operation
    return (number_);
}
void Student::setNumber(long number) {    // operation
    number_ = number;
}
Student:: Student() {    // Default Ctor
    number_=0;
}
Student::Student(long number) {    // Ctor med en inparam.
    number_ = number;
}
Student::~Student( ) {          //Default Dtor
};
```

Testning: Exempel 1: omskrivning 1

```
// Testnin: Student_main.cxx (Student_main.cpp)
#include <iostream>
#include "Student.hxx"
void main()
{
    Student a;    //Student( ) anropas
    Student b(0); // Student(long ) anropas
    //Student b=Student(0);    //samma som objekt b ovan
    Student *c=new Student(0); // Student(long ) anropas
    a.setNumber(10);
    c->setNumber(30);
    cout<<a.getNumber()<<endl;
    cout<<c->getNumber()<<endl;
    delete c;
}
```

Klassdefinition: Exempel 1: omskrivning 2

//Klassdefinition: Student.hxx (Student.h)

```
#ifndef STUDENT_H
#define STUDENT_H
```

//Include:er kommer här!

```
class Student {
```

```
public:
```

```
    Student(long number=0) : number_(number){ } // dekl. & def. av Ctor
                                     //med en imparam. som ersätter de två tidigare!
```

```
    Student(const Student& Acopy): number_(Acopy.number_){ } //Copy Ctor.
```

```
~Student( ){ } // deklaration & def. av default Dtor
```

```
    long getNumber( ) const; // Accessor/Query
```

```
    void setNumber(long); // mutator /Modifier
```

```
private:
```

```
    long number_; // attribut(datamedlem)
```

```
}; // semikolon
```

```
#endif
```

KlassImplementaion:Exempel 1:omskrivning 2

// Implementering av medlemsfunktioner : **Student.cxx**

#include "student.hxx"

long **Student::**getNumber() **const** { //operation

return (number_);

}

void **Student::**setNumber(long number) { // operation

number_ = number;

}

Testning: Exempel 1: omskrivning 2

```
// Test: Student_main.cxx
#include <iostream>
#include "Student.hxx"
void main()
{
    Student a;    //Student( long =0) anropas
    Student b(20); // Student(long =0) anropas
    //Student b=Student(20); //samma som objekt b ovan
    Student *c=new Student( ); // Student(long=0 ) anropas
    Student d=b;    //Student(const Student&) anropas
    Student* e= new Student(b); //Student(const Student&) anropas
    Student f;
    f=a;           //Tilldelningsoperatören anropas
    Student g=2; // Student(long=0 ) anropas //typomvandling från int till
                //Student (Detta sker automatiskt=implicit)
    delete c , e;
}
```

Ett exempel: när & varför behövs Copy Ctor:

```
#include <iostream>
#include <new>
#include <cstdlib>
#include <iomanip>
using namespace std;
class array {
int *p;
int size;
public:
array(int sz) { try { p = new int[sz]; }
                catch (bad_alloc xa) {
                    cout << "Allocation Failure"<<endl;
                    exit(EXIT_FAILURE);
                }
                size = sz;
            }
~array() { delete [] p; }
```

Djup(Deep) vs. Grund (Shallow) kopiering

```
class Object
{
public:
    ~Object(){ delete [] data;}
    int id;
    string name;
    char *data;
};

Object* CloneObject(Object *in)
{
    Object *p = new Object;

    memcpy(p, in, sizeof(Object));
    return p;
}
```

```
void main(void)
{
    char * DataBlock = new char[230];

    Object *object1 = new Object;
    object1->id  = 1;
    object1->data = DataBlock;
    object1->name="NumberOne";

    Object *object2 = CloneObject(object1);
    delete object1;

    // Big Problem:
    delete object2;
}
```


Ett exempel för: varför behövs Copy Ctor:

```
// copy constructor
array(const array &a);
void put(int i, int j) { if ( i>=0 && i<size ) p[i] = j; }
int get(int i) { return p[i]; }
};

// Copy Constructor
array::array(const array &a) {
    try { p = new int[a.size]; }
    catch (bad_alloc xa) {
        cout << "Allocation Failure"<<endl;
        exit(EXIT_FAILURE);
    }
    for(int i=0; i<a.size; i++)
        p[i] = a.p[i];
}
```

Ett exempel: när & varför behövs Copy Ctor:

```
int main()
{
    array num(10);
    for(int i=0; i<10; i++) num.put(i, i);
    for(int i=9; i>=0; i--)
        cout << num.get(i)<<setw(3);
    cout <<endl;
    // create another array and initialize with num
    array x(num); // invokes copy constructor
    for(int i=0; i<10; i++) cout<<setw(3)<<x.get(i);
    cout<<endl;
    return 0;
}
```

Kom ihåg att Copy Ctor anropas endast för initieringar.

Fråga: kräver dessa Copy Ctor?

```
array a(10);
array b(10);
b=a;
```

Övning:

1) Skriv Konstruktor, Destruktor och Copy Konstruktor för Stack-klassen i Exempel2.

Övning:

Skriv Konstruktör, Destruktor och Copy Konstuktur för Stack-klassen i Exempel2.

Konstruktör: → **stack();**
`stack::stack() { count = 0; /* Zero the stack */ }`

Destruktor: → **~stack();**
`stack::~~stack() {
 if (count != 0)
 std::cerr << "Error: Destroying a nonempty stack"<<endl;
}`

Copy Konstuktur: → **stack(const stack& old_stack);**
`stack::stack(const stack& old_stack) {
 for(int i = 0; i < old_stack.count; ++i) { data[i] = old_stack.data[i]; }
 count = old_stack.count;
}`

Inline funktioner

Headerfilen Student.hxx skall inte innehålla definition av medlemsfunktioner, utan endast klassdefinition. Ett viktigt undantag är dock inline-funktioner som MÅSTE definieras i headerfiler, oavsett om dessa inline funktioner är definierade i klassdefinitionen eller separat. OBS!: Medlemsfunktioner som definieras inne i klassdefinitionen betraktas redan som inline-funktioner.

```
inline long Student::getNumber() const {  
    return number_;  
};
```

Skall stå i Student.hxx

Initiering av datamedlemmar:

Initieringar **bör** utföras utifrån deklarationsordningen. Undvik användning av attribut i initieringsuttryck:

```
class A {  
public:  
    A(int s);  
private:  
    int i_;  
    int j_;  
};
```

```
A::A(int s) : j_(s), i_(j_) {} // j först?
```

Explicit konstruerare:

Satsen **Student g=2;** sa vi att anropar konstruerare med ett argument som också kallas för en typomvandlingskonstruktor. Denna omvandling från **int** till **Student** sker automatiskt och det kan skapa problem ibland.

Faktum är att en implicit typomvandling som denna stänger av kompilatorns goda typ-checkningsegenskap som annars skulle säga att dessa typer (Student och int) inte är av samma typ. Därför i de flesta fall bör en sådan konstruerare deklarerars som explicit så att satsen ovan skall inte vara möjlig.

```
class Student {  
    public:  
        explicit Student(long);  
    ...  
    private:  
        long number_;  
};
```

Nu satsen **Student g=2;** BÖR generera fel, men satsen:
Student g(2); är korrekt.

THIS pekare:

Antag att klassen Student har en medlemsfunktion **g()** .

Låt oss se hur **g()** anropas:

1) Skapa ett objekt av typen Student: **Student m(4);**

2) Anropa **g** för objektet **m** med: **m.g();**

- Om vi vill referera till objektet **m** inne i **g()** eller om **g()** gör någon tillståndsändring på objektet **m** och vi till slut vill returnera det nya förändrade objektet **m**. Då måste **g()** utnyttja den konstanta pekaren **this**.

THIS pekare:

this är en konstant pekare till "detta" objekt. "detta" objekt är det objekt som anropar. Man kan inte (på grund av konst-heten) tilldela till **this-pekaren**. För vår Student-klass har denna **this-pekare** formen:

Student *const this;

Denna pekare används också av kompilatorn för att försäkra att en **const medlemsfunktion** inte försöker ändra attributen. Därför för en const medlemsfunktion ser **this-pekare** ut som: **const Student* const this;**

```
inline long Student::getNumber() const {    //operation
    return (number_);
}
```

Är samma som nedan:


```
inline long Student::getNumber() const {    //operation
    return (this->number_);
}
```

Mer om this –pekare kommer senare.

THIS pekare:

```
class Employee {  
    public:  
        ....  
        void setSalary(double sal){ salary = sal; }  
    private:  
        ....  
        int salary;  
        ....  
};
```

```
int main() {  
    ....  
    Employee programmer;  
    Employee professor;  
    professor.setSalary(70000);  
    programmer.setSalary(10000);  
    ....  
}
```



```
void setSalary(Employee *this, int sal)  
{  
    this->salary = sal;  
}
```

THIS pekare:

```
class Point {  
    public:  
        void setX(int x) { _x = x; }  
        void setY(int y) { _y = y; }  
        void doubleMe() { _x *= 2; _y *= 2; }  
    private:  
        int _x;  
        int _y;  
};
```

För att använda denna klass, kan vi skriva kod som:

```
Point lower;  
lower.setX(20);  
lower.setY(30);  
lower.doubleMe();
```

Men vore det inte lite mer naturligt att skriva på detta sätt:

```
lower.setX(20).setY(30).doubleMe(); //kanske, eller hur?
```

Ovanstående sats är samma som:

```
(( lower.setX(20) ).setY(30) ).doubleMe();
```

THIS pekare: Concatenating Calls:

så här:

En omskrivning av föregående exempel:

```
class Point {  
    public:  
        Point& setX(int x) { _x = x; return *this; }  
        Point& setY(int y) { _y = y; return *this; }  
        Point& doubleMe() { _x *= 2; _y *= 2; return *this; }  
    private:  
        int _x;  
        int _y;  
};
```

THIS pekare: Concatenating Calls:

Ännu en omskrivning av föregående exempel:

```
class Point {  
    public:  
        Point& setX(int x) { this->x = x;  
            // "this" pekaren tillåter access till klass-medlemmar  
            return *this;  
        }  
        Point& setY(int y) { this->y = y; return *this; }  
        Point& doubleMe() { x *= 2; y *= 2;  
            // x och y refererar till klass-medlemmar  
            return *this;  
        }  
    private:  
        int x;  
        int y;  
};
```

THIS pekare: Self Identification

```
class Employee {  
    public:  
        Employee(char employeename[]);  
        Employee& operator=(const Employee &);  
        ....  
    private:  
        char *name;  
};
```

```
Employee::Employee(char employeename[]) {  
    name = new char[sizeof(employeename) + 1];  
    strcpy(name,employeename);  
}  
Employee& Employee::operator=(const Employee& rhs) {  
    delete [] name;  
    name = new char[sizeof(rhs.name) + 1];  
    strcpy(name,rhs.name);  
    return *this;  
}
```

THIS pekare: Self Identification

```
class Employee {  
    public:  
        Employee(char employee_name[]);  
        Employee& operator=(const Employee &);  
        ....  
    private:  
        char *name;  
};
```

```
Employee::Employee(char employee_name[]) {  
    name = new char[sizeof(employee_name) + 1];  
    strcpy(name, employee_name);  
}  
Employee& Employee::operator=(const Employee& rhs) {  
    delete [] name;  
    name = new char[sizeof(rhs.name) + 1];  
    strcpy(name, rhs.name);  
    return *this;  
}
```

THIS pekare: Self Identification

```
int main() {  
    .....  
    Employee a("John");  
    Employee b("Jokkapala");  
    ....  
    a = b;  
    .....  
}
```

Vad händer om vi (kanske av misstag) skriver :

a = a;

THIS pekare: Self Identification

Vad händer om vi (kanske av misstag) skriver : `a = a;`

dvs. ett Employee objekt tilldelas till sig själv.

Lite osannolikt, men om objekten var lagrade i ett fält eller en lista eller var refererade med pekare, kan detta hända.

Första steget i tilldelnings-Operatörn bufferten för **name** frigörs, det går bra, men i nästa steg med **sizeof** operationen blir det boom boom. **"rhs.name"** har tagits bort och därför kommer programmet att dö. Men vi kan förhindra detta med "this" pekaren:

```
Employee& Employee::operator=(const Employee& rhs) {  
    if (this != &rhs) {  
        delete [] name;  
        name = new char[sizeof(rhs.name) + 1];  
        strncpy_s(name, sizeof(rhs.name), rhs.name, sizeof(name));  
    }  
    return *this;  
}
```

Klassvariabler= statiska medlemmar:

Om en datamedlem är gemensam för alla objekt som tillhör en viss klass, denna medlem kallas för en **statisk datamedlem**.

En sådan datamedlem finns bara i en enda instans som delas av alla objekt av den klassen.

Exempel på en statisk datamedlem kan vara en variabel som håller reda på hur många objekt av en viss klass som finns för närvarande.

Ett annat är till exempel datamedlemmen universitet för alla studenter.

Generell syntax: **static typ namn;** //deklaration

Definitionen måste läggas utanför klassen:

typ klassnamn::namn=initieringsvärde;

Den statiska datamedlemmen existerar först när den definieras.

Klassdefinition: Exempel 1: omskrivning 3

```
#ifndef STUDENT_H
#define STUDENT_H
class Student {
public:
    Student( );
    Student(const Student& Acopy): number_(Acopy.number_){ }
    ~Student( ){ };
    long getNumber( ) const;
    void setNumber(long);
    static string getUniversity( ) { return (university_); } //behandlar endast statiska
                                                                //medlemmar.

    static long currentNumberUsed( ) { return currentNumber_; }
private:
    long number_;
    static string university_; //statisk datamedlem , deklaration
    static long currentNumber_; //statisk datamedlem , deklaration
}; // semikolon

#endif
```

Klassdefinition:Exempel 1: omskrivning 3

```
// Definition av medlemsfunktioner : Student.cxx
#include "student.hxx "
//definition av de statiska datamedlemmarna
string Student::university_(" Mid Sweden ");
long Student::currentNumber_=0;

long Student::getNumber() const { return (number_); }

void Student::setNumber(long number) { number_ =
    number; }

Student::Student( ) { number_ = currentNumber_++; }
```

Enumrering

Ibland behöver vi uppräkningsbara typer i en klass, speciellt att dessa kan ersätta konstanta statiska (numeriska) datamedlemmar.

```
class Student {  
    public:  
        enum Gender { Undefined, M, F };  
        Student(); //deklaration av default konstruerare  
        Student (string, long, Gender);  
        Gender getGender() { return (Gen_); }  
        .....  
    private:  
        string name_;  
        Gender Gen_;  
        .....  
}; // semikolon  
#endif
```

Enumrering (forts.)

```
Student::Student() { // Default Ctor
    number_ = currentNumber_++;
    name_ = " ";
    Gen_ = Undefined; // Gen_ = Student::Undefined;
}
```

```
Student::Student(string name, long number, Gender g) {
    name_ = name;
    Gen_ = g;
    number_ = number < 0 ? -1 : number;
}
```

```
//andvändning
Student a("kaisa", 100, Student::F );
Student b;
cout<<b.getGender()<<endl;
cout<<b.getUniversity()<<endl; //eller cout<<Student::getUniversity()<<endl;
```

Privata konstruktorer

Användning av en konstruktor kan begränsas till de privata medlemsfunktionerna genom att göra den till en privat medlemsfunktion:

```
class Student {  
public:  
    explicit Student(long); // publik konstruktor  
    ...  
private:  
    Student();              // Endast för medlemmar  
    long number_;  
};
```

Konstanta datamedlemmar

Man kan inte initiera konstanta attribut inne i en klassdefinition.
Dessa **MÅSTE** initieras m.h.a en konstruktor:

```
class Student {  
public:  
    explicit Student(long, int);  
    const int sin;          // Inga initieringar här  
    ...  
private:  
    long number_;  
};  
  
Student::Student(long number, int s) : sin(s) {  
    number_ = number;  
}
```


Funktioner med Referensparametrar

Modifiera the värde som pekaren pekar på:

// ändra värdet om det är mindre än 100.

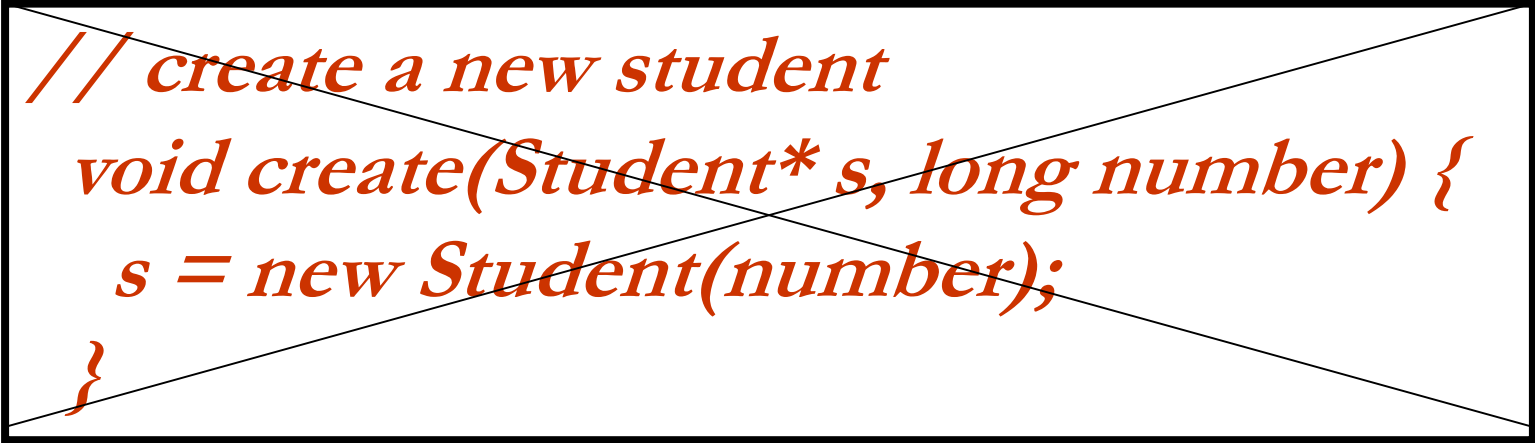
```
void modifyNumber(Student* s) {  
    long number = s->number();  
    if(number < 100)  
        s->setNumber(number + 100);  
}
```

Referensparameter:

```
void modifyNumber(Student& s) {  
    long number = s.number();  
    if(number < 100)  
        s.setNumber(number + 100);  
}
```

// "." inte "->"

Funktioner med Referensparametrar(forts.)



```
// create a new student  
void create(Student* s, long number) {  
    s = new Student(number);  
}
```

```
void create(Student*& s, long number) {  
    s = new Student(number);  
}
```

```
Student* create(long number) {  
    return new Student(number);  
}
```

Funktioner som Returnerar Referenser

En funktion kan returnera en referenstyp, om så, ett funktionsanrop är en L-value:

```
int& foo() {  
    static int x = 0;    //ett alternativ till en global variabel  
    return x;  
}
```

```
cout << foo() << endl;    //skriver ut 0  
foo() += 3;  
cout << foo() << endl;    // skriver ut 3
```

```
int a[5];  
int& index(int x[], int i) { //ger läs/skriv access till i:e elementet av x  
    return x[i];  
}  
cin >> index(a, 1);
```

Funktioner som Returnerar const Referenser

Funktioner kan också skrivas så att de ger endast läs-access, genom att returnera en const referens:

```
const int& get(const int x[], int i) { return x[i]; }
```

```
cout << get(a,2);
```

```
cin >> get(a,1); // kan inte ändra värdet
```

Så när skall jag använda funktioner som returnerar referenser:

Till exempel:

- När stora objekt skall returneras eller funktionen skall användas som l-value, vilket är fallet med de flesta operatorer som +, -, * etc.

Medlemsfunktioner som Returnerar Referenser

```
//Integer.hxx
```

```
#ifndef INTEGER_H
```

```
#define INTEGER_H
```

```
class Integer {
```

```
    public:
```

```
        Integer(int = 0);
```

```
        int get() const;
```

```
        Integer& set(int);
```

```
        Integer& add(const Integer&);
```

```
    private:
```

```
        int value_;
```

```
};
```

```
#endif
```

Medlemsfunktioner som Returnerar Referenser

// Medlemsfunktion Implementation av Integer klassen // **Integer.cxx**

```
#include "Integer.hxx"
```

```
Integer::Integer(int i) { value_=i; }
```

```
int Integer::get( ) const { return value_; }
```

```
Integer& Integer::set(int n) {
```

```
    value_ = n;
```

```
    return *this;
```

```
}
```

```
Integer& Integer::add(const Integer& i) {
```

```
    value_ += i.value_;
```

```
    return *this;
```

```
}
```

Returnera nya objekt

Om man vill ha ett nytt objekt av en klass, en medlemsfunktion kan returnera ett nytt objekt (INTE en referens) av den klassen:

```
class Integer {  
    public:  
        Integer(int = 0);  
        Integer clone() const;  
    private:  
        int value_;  
};
```

```
Integer Integer::clone() const {  
    Integer res(value_);    //stack  
    return res;  
}
```

//Bättre alternativ

```
Integer& Integer::clone() {  
    Integer* copy = new Integer(value_); // heap  
    return *copy;  
}
```

Medlemsfunktioner som Returnerar Referenser

```
//Integer_main.cxx
#include "Integer.hxx"
#include <iostream>
using namespace std;
void main ( ) {
```

```
    Integer i(5);    Integer j(i);    // default kopieringskonstruktor
```

```
    cout << "i = " << i.get() << "; j = " << j.get() << endl;
```

```
    j = i.set(2).add(3);
```

```
    cout << "i = " << i.get() << "; j = " << j.get() << endl;
```

```
    i = i.add(j).add(j); // kedje addition: (i.add(j)).add(j)
```

```
    cout << "i = " << i.get() << "; j = " << j.get() << endl;
```

```
    Integer k = i.add(3);    Integer m = i.add(k);
```

```
    cout << "i = " << i.get() << "; j = " << j.get() << endl;
```

```
    cout << "k = " << k.get() << "; m = " << m.get() << endl;
```

```
    Integer b=i.clone();    cout<<"b.get(): "<<b.get()<<endl; }
```

Utskrift:

i = 5; j = 5

i = 5; j = 5

i = 15; j = 5

i = 36; j = 5

k = 18; m = 36

b.get(): 36

Nästlade objekt

Ett objekt av en klass med attribut av en annan klass-typ (attributet är inte pekare eller referens) sägs innehålla **nästlade objekt** INTE referenser till ett annat objekt.

Ex.: Till klassen student kan du definiera en annan klass som beskriver ett konto för studenten med ett nästlat objekt som representerar studenten.

Se exemplet på nästa bild.

Nästlade objekt

```
class AccountForStudent {
public:
    AccountForStudent(Student, double balance);
    AccountForStudent(long number, string name,
        double balance);
    ...
private:
    Student stud_;           // nästlat objekt
    double balance_;
};
AccountForStudent::AccountForStudent(Student s, double balance):
    stud_(s), balance_(balance) { }

AccountForStudent::AccountForStudent(
    long number, string name, double balance) :
    stud_(number, name), balance_(balance) { }
```

Nästlade Klasser

```
class Surround {  
    public:  
        class FirstWithin {  
            int d_variable;  
            public:  
                FirstWithin();  
                int var() const { return d_variable; }  
        };  
    private:  
        class SecondWithin {  
            int d_variable;  
            public:  
                SecondWithin();  
                int var() const { return d_variable; }  
        };  
};
```

Vänner: Friendfunktioner

är icke-medlemsfunktioner med fullständig insyn i en klass.

```
#ifndef _VARA_H_
#define _VARA_H_
class Vara {
public:
    Vara(int antal = 0, double pris = 0.0) : antal_(antal), apris(pris) { }
    friend double TotalPris( Vara &i);
private:
    int antal_;
    double apris_;
    //friend double TotalPris( Vara &i);
};
double TotalPrice(Vara &i) { return i.antal_ * i.apris_; }
#endif
```

Vänner: Friendfunktioner

```
#include <iostream>
#include "Vara.h"
using namespace std;
int main(int argc, char* argv[])
{
    Vara item(12, 0.99);
    cout << "Total Pris: " << TotalPris(item) << endl;
    return 0;
}
```

Vänner kan vara:

- En annan klass
- Icke-medlemsfunktion (vanlig funktion)
- En medlemsfunktion av en annan klass

Vanligaste användningen är operatorer

- om *vänstra* operanden inte är “vår klass”

Mer om operatorer senare.

Vänner kan vara (forts.):

Klassen C kommer före

```
class B
```

```
{
```

```
    friend void C::medlemsfunktion(const B &);
```

```
// .....
```

```
};
```

```
class A
```

```
{
```

```
    friend class B;
```

```
    // etc
```

```
    private:
```

```
    // klass A's datamedlemmar.
```

```
};
```

Ömsesidig vänskap!:

Om två klasser vill ge ömsesidig tillgång till sina privata medlemmar, den ena klassen kan deklareraras först. Detta kallas "forward declaration" eller "forward reference".

class A; 

```
class B
{
    friend class A;
    //.....
};
```

```
class A
{
    friend class B;
    // etc
    private:
        // klass A's
        datamedlemmar.
};
```


Ömsesidig vänskap!:

```
#include <iostream>

class CSquare; //forward declaration
class CRectangle {
    int width, height;
public:
    int area ( ) { return (width * height); }
    void convert (CSquare a);
};

class CSquare {
    int side;
public:
    void set_side (int a) { side=a; }
    friend class CRectangle;
};
```

```
void CRectangle::convert (CSquare a)
{
    width = a.side; height = a.side;
}

void main () {
    CSquare sqr;
    CRectangle rect;
    sqr.set_side(4);
    rect.convert(sqr);
    cout << rect.area();
}
```

Preprocessor direktiv: Mer C än C++

- Preprocessor:n innehåller följande direktiv:
- `#define` `#elif` `#else` `#endif` `#error` `#if`
`#ifdef` `#ifndef` `#include` `#line` `#pragma` `#undef`
- En preprocessor direktiv börjar alltid med `#`
- Två direktiv kan inte finnas på samma rad, tex:
`#include <iostream> #include <cstdlib>`
fungerar inte.

Preprocessor direktiv: #define

- Syntax: **#define** *macro-name char-sequence*
- Definierar en identifierare **macro-name** och en **symbolsekvens** som skall ersättas av identifeieraren **macro-name**

■ Ex:

```
#define ONE 1
```

```
#define TWO ONE+ONE
```

```
#define THREE ONE+TWO
```

```
cout<< ONE<<endl<<TWO<<endl<< THREE<<endl;
```

```
#define E_MS "standard error on input\n "
```

```
/* ... */
```

```
cout<<E_MS;
```

Preprocessor direktiv: #define

```
#define LONG_STRING "this is a very long string, \  
    use backslash if the string is longer than one line"
```

```
#define MAX_SIZE 100
```

```
int balance[MAX_SIZE];
```

```
for(int i=0; i<MAX_SIZE; i++) balance[i]=rand()%35 +1;
```

- Nedan är en funktionsmacro

```
#define ABS(a) ( (a)<0 ? -(a) : (a) )
```

```
cout<<"abs of -1 and 1:"<<ABS(1)<<endl;
```

Preprocessor direktiv: #if, #else och #endif

- Syntax för **#if** och **#else**:
#if *constant-expression*
statement sequence
#else
statement sequence
#endif

- Ex:

```
#define MAX 100
```

```
#if MAX>99
```

```
    cout<<"Compiled for array greater than 99."<<endl;
```

```
#else
```

```
    cout<<"Compiled for small array."<<endl;
```

```
#endif
```

Preprocessor direktiv: #elif och #endif

Syntax för **#elif**:

```
#if expression
    statement sequence
#elif expression 1
    statement sequence
#elif expression 2
    statement sequence
#elif expression 3
    statement sequence
#elif expression 4
    .
    .
    .
#elif expression N
    statement sequence
#endif
```

Preprocessor direktiv: #elif och #endif

EX :

```
#define US 0
#define ENGLAND 1
#define FRANCE 2
#define SWEDEN 3
#define ACTIVE_COUNTRY SWEDEN
#if ACTIVE_COUNTRY == SWEDEN
char currency[] = "kronor";
#elif ACTIVE_COUNTRY == ENGLAND
char currency[] = "pound";
#elif ACTIVE_COUNTRY == US
char currency[] = "dollar";
#else
char currency[] = "franc";
#endif
```

Preprocessor direktiv: #elif och #endif

- C++ tillåter upp till 256 nästlade #if och #elif.
- Isf. varje #endif, #else, eller #elif associeras med närmaste #if eller #elif.

- EX:

```
#if MAX>100
    #if SERIAL_VERSION
        int port=198;
    #elif
        int port=200;
    #endif
#else
    char out_buffer[100];
#endif
```


Preprocessor direktiv: #ifdef och #ifndef

- En annan metod för villkorlig kompilering
- **Generella syntaxen för #ifdef:**

`#ifdef macro-name`

`statement sequence`

`#endif`

- **Generella syntaxen för #ifndef:**

`#ifndef macro-name`

`statement sequence`

`#endif`

EX:

```
#define TED 10
```

```
#ifdef TED
```

```
    cout<<"Hi Ted\n";
```

```
#else
```

```
    cout<<"Hi anyone\n";
```

```
#endif
```

```
#ifndef RALPH
```

```
    cout<< "RALPH not defined\n";
```

```
#endif
```

Preprocessor direktiv: #undef

- Tar bort en tidigare definierad macro
- Generella syntaxen för #undef: **#undef** *macro-name*

- **EX:**

```
#define LEN 100
```

```
#define WIDTH 100
```

```
    char array[LEN][WIDTH];
```

```
#undef LEN
```

```
#undef WIDTH
```

```
/* Här både LEN och WIDTH är odefinierade. */
```

Användning av: `defined`

- Ett annat sätt för att se om en macro är definierad är: `defined`

- **Syntax:** `defined macro-name`

- *Ex1:*

```
#if defined MYFILE
```

```
/*eller */
```

```
#ifdef MYFILE
```

- *Ex2:*

```
#if !defined DEBUG
```

```
cout<<"Final version!\n";
```

```
#endif
```

Preprocessor direktiv: #line

- Direktiven **#line** ändrar innehållet i **__LINE__** och **__FILE__** som är fördefinierade identifierare i kompilatorn. **__LINE__** innehåller radnumret i den nuvarande kompilerade kodraden.
__FILE__ är en sträng som innehåller namnet till den fil som kompileras
- Den generella formen för **#line** är: **#line *number* [*filename*]**
där *number* är ett positivt tal och som blir det nya värdet för **__LINE__** och *filename* är en giltig fil-identifierare som blir det nya värdet för **__FILE__**
- **#line** används normalt för debug.

Preprocessor direktiv: #line

- Tex. Följande kod anger att rad-räkning skall börja med 100. **cout** skriver ut 102, eftersom det är 3:e raden i programmet efter **#line 100** satsen.

```
#include <iostream>
using namespace std;
#line 100 /* reset the line counter */
int main( ) /* line 100 */
{ /* line 101 */
cout<< __LINE__ <<endl; /* line 102 */
return 0;
}
```

Preprocessor direktiv: #pragma

- **#pragma** är en implementations-beroende direktiv som möjliggör olika instruktioner kan ges till kompilatorn.
- Tex. En kompilator kan ha en växel (option) som stöder s.k. ”program execution tracing”, då kan man ange den växeln med **#pragma**
- **#pragma warning (disable : 4996)**
VS. C++ 2K5 avråder användning av funktioner som strcpy av säkerhetsskäl, detta stänger av den varning som genereras.

Preprocessor operatorer: # och

- Dessa operatorer används tillsammans med **#define**
- **#** operatorn, som generellt kallas för: ”*stringize*” operatorn, omvandlar följande argument till (quoted) sträng.
- Tex. i nedanstående kod:

```
#define mkstr(s) #s
```

```
cout<<mkstr(I like C++)<<endl;
```

```
cout satsen omvandlas till: cout<<”I like C++”<<endl;
```

Preprocessor operatorer: # och

- Dessa operatorer används tillsammans med **#define**
- **##** operatorn, som generellt kallas för: ”*pasting*” operatorn, slår samman två tokens.

- Tex. i nnedanstående kod:

```
#define concat(a, b) a ## b
```

```
int xy = 10;
```

```
cout<< concat(x, y)<<endl;
```

- `cout` satsen omvandlas till: **cout<<xy<<endl;**

```
#define writeout(a,b) a ## b
```

```
writeout(c,out) << "test";
```

- Omvandlas till: **cout<<"test";**

Fördefinierade preprocessor direktiv

C++ har 6 fördefinierade macro-namn, 5 av dessa definieras av språket C, dessa är:

__LINE__

__FILE__

__DATE__

__TIME__

__STDC__

__cplusplus



De innehåller nuvarande radnummer och filnamnet till programmet under kompilering. **Se bilden om #line**

Fördefinierade preprocessor direktiv

- `__DATE__` macron innehåller en sträng av formen *month/day/year* som anger översättningsdatumet av källkoden till objekt-kod.
- `__TIME__` macron innehåller programmets kompileringstid. Den är en sträng av formen *hour.minute.second*.
- Betydelsen av `__STDC__` är implementationsberoende. Generellt, om `__STDC__` är definierad, kompilatorn accepterar endast standard C/C++ kod med inga andra icke-standard utökningar.
- En kompilator som stöder standard C++, definierar `__cplusplus` som ett 6-siffrigt värde. De övriga (som inte stöder) använder ett 5-siffrigt (eller mindre) värdet.

assert macron med #define

```
#define assert(test) \
{\
    if (!(test)) \
    {\
        cout << "Assertion: \"\" << #test << "\" on line \"\
            << __LINE__ <<endl<<" in file \" << __FILE__ << "."<<endl\
            <<"This error happened in:"<<__DATE__<< "."<<endl\
            <<"at:"<<__TIME__<< "."<<endl;\
        ::exit(-1);\
    }\
} //backslash på alla rader förutom den sista.
```

- Assertion: "0" on line 34
- in file c:\users\naymal\desktop\test\test\test\hof.cpp.
- This error happened in: Sep 5 2008.
- at:03:40:05.

Preprocessor direktiv: #error

- Syntax: **#error** *error-message*

```
#ifndef __cplusplus
```

```
#error A C++ compiler is required!
```

```
#endif
```

Operator-överlagring

Nayeb.Maleki@miun.se

Operator-överlagring:

- Antag att vi har klassen definitionen:

```
class Student {  
    public:  
        explicit Student(long number, const char* name); // dekl. av en Ctor.  
        ~Student( );  
        long getNumber( ) const; // Accessor/Query  
        void setNumber(long); // mutator /Modifier  
    private:  
        long number_; // attribut(datamedlem)  
        const int sin_;  
        char *name_  
        Student();  
};
```

- Och att vi av ngn. anledning vill skriva koden:

```
Student s1(10), s2(1);
```

```
if(s1>s2) cout<<"S1 har högre \"id\" nummer."<<endl;
```

Operator-överlagring (forts.):

Problemet här är att `s1` och `s2` är inte objekt av ngn. av de primitiva (fördefinierade) data-typer kompilatorn känner till och för vilka finns en definierad betydelse för jämförelsen `s1 > s2`, därför `s1 > s2` kan inte gå igenom kompilatorn.

För att ge kompilatorn en riktlinje för hantering av objekt av egen-definierade typer MÅSTE vi överlagra operatören `>` och därmed ge en mening till `s1 > s2`

En överlagrad operator är i själva verket en funktion med:

- Ett namn med den generella syntaxen: `operator[operator-symbol]`
Tex. `operator >`
- En returtyp
- Ett eller två (förutom den ternära villkors-operatören `?:`) argument.

Exempel:

```
bool operator > (const Student& rhs);  
bool operator < (const Student& lhs, const Student& rhs);
```

Operator-överlagring (forts.):

Exempel: dekl. : `bool operator > (const Student& rhs);`

`Student s1(10), s2(1);`

`if(s1 > s2)`

.....

`(kontrollerande operand=lhs) operator (argument operand=rhs)`

s1

>

s2

(**lhs** står för **L**eft **H**and **S**ide, **rhs** står för **R**ight **H**and **S**ide)

Exempel: definition:

`bool Student::operator > (const Student& rhs)`

`{`

`if(number_ > rhs.number_)`

`return true;`

`else`

`return false;`

`}`

Observera att vi behöver inte skicka med **lhs**, som är **detta objekt** dvs. objektet som vi anropar operatoren för. **Denna operator implementeras därför som medlemsfunktion.**

Operator-överlagring (forts.):

En **operator-ID(symbol)** kan vara en av nedanstående:

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

- Nedanstående operatorer kan överlagras i sina binära och unära former:

+ - * & (till exempel : **+10** och **X+10**)

- Nedanstående operatorer kan **INTE** överlagras:

. .* :: sizeof # ## ?: ..._cast<>

Operator-överlagring (forts.):

- `[] () = ->` kan endast deklarerars som "nonstatic" medlemsfunktioner, detta försäkrar att deras första operand är en Lvalue.
- När du vill överlagra en operator, utgå från den fördefinierade motsvarigheten (kallad: **operator interface**).

Till exempel:

- `==` är för att jämföra två operander **av samma typ** (**`a==b`**). Det skall den överlagrade motsvarigheten vara till för också och inget annat! En sådan operator sägs ha en symetrisk egenskap.
- Operator-överlagring sker endast för användar-definierade typer. **Minst ett argument måste vara användar-definierad.**

Operator-överlagring (forts.):

- **Varken** ordningsföljden (precedence) **eller** antal argument för en operator kan ändras. Till exempel: **&&** skall alltid anta två argument och för överlagrade **<** , **/** beräkningen $t2 < t1 / t2$ alltid behandlas som $t2 < (t1 / t2)$
- **Vi kan inte hitta på egna operatorer:**
void operator @ (int); //illegal, @ is not a built-in operator or a type name
- Tvärtemot vanliga funktioner, överlagrade operatorer kan inte deklareras med en default parameter (gäller inte **()**)

Student & operator += (const Student & d = Student());

//error, default argument

3 sätt att överlagra operatorer

- Som en *icke-medlemsfunktion* till en klass
 - Något ineffektivt, eftersom de har inte access till klassens privata medlemmar → måste använda accessor eller mutator metoder.
 - Har så många argument som operatören kräver (unary, binary, etc).
- Som en medlemsfunktion till en klass:
 - Har access till privata medlemmar.
 - Har *ett mindre* argument än det antal operander operatören som överlagras kräver.
 - Har ett implicit argument (LHS operanden==objektet själv).
- Som *friend* till en klass

Operator-överlagring (forts.):

- Som en *icke-medlemsfunktion* till en klass:

- Student.h

- ```
class Student {
 public:

 private:

};

bool operator >(const Student &lhs , const Student &rhs);
```

- Student.cpp

```
bool operator > (const Student& lhs, const Student& rhs)
{
 if(lhs.getNumber() > rhs.getNumber()) return true;
 else
 return false;
}
```

- Student\_main.cpp

- ```
if( s1 > s2)  ....
```

Operator-överlagring (forts.):

- Som en medlemsfunktion till en klass:

- **Student.h**

- class Student {
 public:

 bool operator > (const Student& rhs);
 private:

};

- **Student.cpp**

- bool Student::operator > (const Student& rhs)**
 {
 if(number_ > rhs.number_) return true;
 else
 return false;
 }

- **Student_main.cpp**

- **if(s1 > s2)**

Operator-överlagring (forts.):

- Som *friend* till en klass:

- **Student.h**

- class Student {
public:
.....
friend bool operator >(const Student &lhs , const Student &);
private:
.....
};

- **Student.cpp**

- bool operator > (const Student &lhs , const Student& rhs)**
{
if(lhs.number_> rhs.number_) return true;
else
return false;
}

- **Student_main.cpp**

- **if(s1 > s2)**

Operator-överlagring (forts.):

- Med operatorn `>` överlagrad, vi kan inte skriva:

- `s1 < s2`

- `bool operator < (const Student& rhs);`

- `bool Student::operator < (const Student& rhs)`

- `{`

- `if(number_ < rhs.number_)`

- `return true;`

- `else`

- `return false;`

- `}`

Operator-överlagring (forts.):

- `bool operator == (const Student& rhs);`
- `bool Student::operator == (const Student& rhs) {
 if(strcmp(name_ , rhs.name_)==0)
 return true;
 else
 return false;
}`
- `if(s1==s2) cout<<"de är lika"<<endl;`

Operator-överlagring (forts.):

```
bool Student::operator >= ( const Student& rhs)
{
    //if(strcmp(name_, rhs.name_) > 0 || strcmp(name_, rhs.name_) == 0)
        if(*this > rhs || *this == rhs)
            return true;
    else
        return false;
}

bool Student::operator <= ( const Student& rhs)
{
    //if(strcmp(name_, rhs.name_) > 0 || strcmp(name_, rhs.name_) == 0)
        if(*this < rhs || *this == rhs)
            return true;
    else
        return false;
}
```

Operator-överlagring (forts.):

- `Student& operator = (const Student& rhs);`
- `Student& Student::operator = (const Student& rhs)`
`{`
 `if(this == &rhs)`
 `return *this;`

 `strcpy_s(name_, sizeof(rhs.name_), rhs.name_);`
 `number_ = rhs.number_;`
 `return *this;`
`}`

Operator-överlagring: ++ och --

- Dessa två operatorer returnerar original-värdet i postfixform och det nya värdet i prefixform.

- **The prefix form:**

```
T& T::operator++() {  
    // perform increment  
    return *this;  
}
```

```
T& T::operator--() {  
    // perform decrement  
    return *this;  
}
```

- **The postfix form: (with a dummy **int** , to separate from prefix form)**

```
T T::operator++(int) {  
    T old( *this );  
    ++*this;  
    return old;  
}
```

```
T T::operator--(int) {  
    T old( *this );    // remember old value  
    --*this;           // - call the prefix version  
    return old;        // - return the old value  
}
```

Operator-överlagring (forts.): försök1

- **friend ostream& operator<< (ostream &, const Student &);**
- **ostream & operator<<(ostream &os, const Student &s) {
 os <<s.name_ <<"\t"<<s.number_<<endl;
 // os <<s.getName()<<"\t"<<s.getNumber()<<endl;
 return os;
}**
- **cout<<s<<endl;**
-

Operator-överlagring (forts.): försök 2

- **friend ostream& operator<< (ostream &, const Student &);**
- **ostream &print(ostream &os) const;**
- **ostream & operator<<(ostream &os, const Student &s) {
 s.print(os);
 return os;
}**
- **ostream & Student::print(ostream &os) const {
 os <<name_ <<"\t"<<number_<<endl;
 return os;
}**
- **cout<<s<<endl;**
-

Operator-överlagring (forts.):

- `friend istream& operator>> (istream &, Student &);`
- `istream & operator>>(istream &is, const Student &s) {
 char name[20];
 long number;
 cout<<"Ange namn: ";
 is>>name;
 cout<<"Ange nummer: ";
 is>>number;
 s.setNumber(number);
 s.setName(name);
 return is;
}`
- `cin>>s;`

Operator-överlagring: Tabeller

Table 1. Operator format i C++

Expression Format	Operator Type	Examples
lhs operator rhs	Binary	a + b, a > b
lhs operator	Unary (Postfix)	a++, a--, a->
operator rhs	Unary (Prefix)	++a, --a, *a
operand1 ? operand2 :	Ternary	a > b ? a : b

operand3

Table 2. Unary operator overloading

Operator	Examples	Operator	Examples
+	+a	~	~a
-	-a	!	!a
*	*a	++	a++, ++a
&	&a	--	a--, --a
->	a->		

Operator-överlagring: Tabeller

Table 3. Binary operator overloading

Operator	Example	Operator	Example
+	a + b	&=	a &= b
-	a - b	=	a = b
*	a * b	<<	a << b
/	a / b	>>	a >> b
%	a % b	>>=	a >>= b
^	a ^ b	<<=	a <<= b
&	a & b	==	a == b
	a b	!=	a != b
=	a = b	<=	a <= b

Operator-överlagring: Tabeller

Table 3. Binary operator overloading (forts.)

Operator	Example	Operator	Example
<	a < b	>=	a >= b
>	a > b	&&	a && b
+=	a += b		a b
-=	a -= b	,	a,b
*=	a *= b	[]	a[b]
/=	a /= b	()	a(b)
%=	a %= b	->*	a->*b
^=	a ^= b		

Operator-överlagring: Tabeller

Table 4. Suggested guidelines for operator overloading

Type	Operators	Implementation
Unary	->	Member function (required)
Binary	= () []	Member function (required)
Unary	+ - * &	Member function
Binary	+ - * / &	Nonmember function
Unary	++ -- ~ !	Member function
Binary	+= -= /= *= >>=	Member function
Binary	^= &= = %= <<=	Member function
Binary	% ^ << >>	Nonmember function
Binary	> < >= <= == !=	Nonmember function
Binary	&& , ->*	Nonmember function

Operator-överlagring: Tabeller

Table 5. Overloaded cast operator formats

Expression	Meaning	Member Function
<code>static_cast<Type>(a)</code>	static cast	<code>a.operator Type ()</code>
<code>(Type)a</code>	traditional cast	<code>a.operator Type ()</code>
<code>Type(a)</code>	functional cast	<code>a.operator Type ()</code>

Operatorer och dess syntax som medlemsfunktioner

Statement	Operator	Syntax
$\bullet a$	$+- * \&! \sim ++ --$	$A::operator \bullet ()$
$a \bullet$	$++ --$	$A::operator \bullet (int)$
$a \bullet b$	$+- * / \% ^$	$operator \bullet (A, B)$
$a \bullet b$	$\& < > == !=$	$operator \bullet (A, B)$
$a \bullet b$	$< = > = < < > > \& \& ,$ $= + = - = * = / = \% = ^ = \& = = <$ $< = > > = []$	$A::operator \bullet (B)$
$a(b, c...)$	$()$	$A::operator () (B, C...)$
$a \rightarrow x$	\rightarrow	$A::operator \rightarrow ()$

Operatorer och dess syntax som globala funktioner

Statement	Operator	Syntax
$\bullet a$	$+- * \&! \sim ++ --$	$\text{operator} \bullet (A)$
$a \bullet$	$++ --$	$\text{operator} \bullet (A, \text{int})$
$a \bullet b$	$+- * /\% ^ <= >= << >> \&\& ,$ $\& <> == !=$	$\text{operator} \bullet (A, B)$