Klasser i C++

```
#include <string>
class Person {
    std::string m_name;
    std::string m_address;
    int m_skonummer;
    public:
        void setName(string const &str);
        void setAddress(string const &str);
        string const &name() const;
        string const &address() const;
};
```

Klassdeklaration = interface
Definition av funktioner i klassen = implementation

Klasser – konstruktor - destruktor

- Två specialfunktioner som har att göra med hur klassen internt fungerar – konstruktor coh destruktor
  - Konstruktor – metod som efter att objektet skapats, men före access av detta
  - Samma namn som klassen, ingen returtyp, kan finnas överladdade konstruktorer

```
class Person {
public:
    Person();           /*Default konstruktor */
    Person(string const &str); /* Överladdad */
}
main() {
  Person p;           /* Default används */
  Person p2(”Axel Eklund”);   /* Överladdad */
}
```

Klasser – konstruktor - destruktor

– Destruktor – metod som exekveras just innan objektet slutar att existera
– Samma namn som klassen, med ”~” framför. Ingen returtyp, inga argument.

```
class Person {
public:
    ~Person();          /* Destruktor */
}
```

Klasser – konstruktor - destruktor

- Vad görs i konstruktorn / destruktorn
  - Initialisering av datafält
  - Allokering/friställning av minne

```
class Person {
Person::Person() {
  m_name = "";
}
Person::Person(string const &str) {
  m_name = str;
}
Person::~Person() {
   m_name="";
}
```

Klasser – konstruktor - destruktor

```
/* mystring.h */
class String {
private:
   unsigned int m_len;     /* Stränglängd */
   char *m_str;         /* Pekare till strängen */
public:
   String() {m_len=0;m_str=0;}/* Default konstruktor */
   String(char const *str);   /* Överladdad konstr.*/
   ~String()            /* Destruktor */
};
/* mystring.cpp */
String::String(char const *str) {
   m_len = strlen(str);
   m_str = malloc(sizeof(char)*(m_len+1)); /* Allokera minne */
   strcpy(m_str,str);
}
String::~String() {
   if (m_str) free(m_str);    /* Frigör allokerat minne */
}
```

Klassmetoder – inline/direkta metoder

- Metoderna kan finnas definierade i klassdeklarationen

```
class String {
public String() {m_len=0;m_str=0;}
}
```

  - Detta innebär att inget funktionanrop skapas, koden inkluderas ”inline”
  - Samma kan skapas med ”inline”-deklarationen, men då måste impl. finns i include-filen

```
class String {
public String();
}
inline String::String() {
  m_len =0; m_str = 0;
}
```

Sammansatta klasser

- Klasser har ofta i sin tur medlemmar som också är klasser → komposition
- Ex. Person-klassen har datafält av typen string, vilken i sin tur är en klass
- När vi har nästade klasser, hur kommer initialiseringen av de olika klasserna att gå till?
  - Kompilatorn skapar kod som initialiserar behövliga klassar, generar kod för anrop av behövliga kontruktorer

Initialisering av sammansatta klasser

- Då ett objekt skapas, kommer samtidigt objekt av underliggande klasser att skapas, och deras konstruktorer kallas
  - Vi kan explicit definiera hur dessa skall initialiseras
  - Också elementära datatyper i strukturer kan initialiseras

```
Person::Person(char const *name, char const
  *address, unsigned skonum)
:
m_name(name),/* string konstruktor */
m_address(address), /*   -”” -  */
m_skonummer(skonum) /* init av unsigned *(
{} /* Ingen initialiseringskod */
```

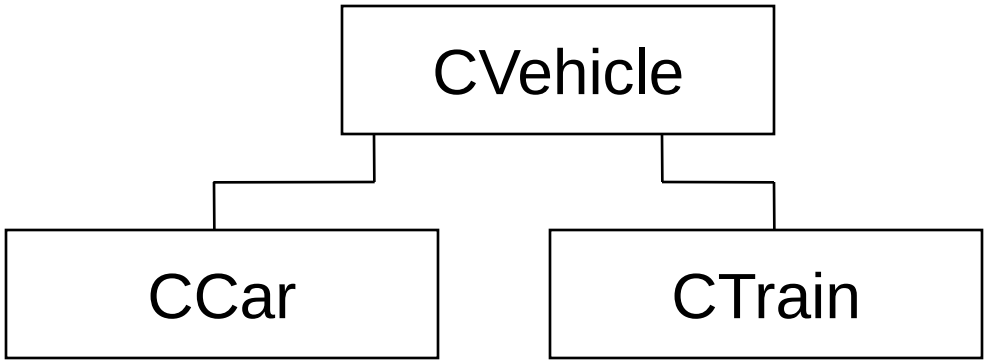
Ärvning

- Objektorienterade språk implementerar i regel ärvning (inheritance). Genom ärvning skapar man en ny klass som ärver förälderns egenskaper

```
class CCar: public CVehicle {  /* Dvs ärver CVehicle */
    unsigned m_speed;
    public:
        CCar();
        CCar(float w, unsigned speed);
        void SetSpeed(unsigned speed);
};
```

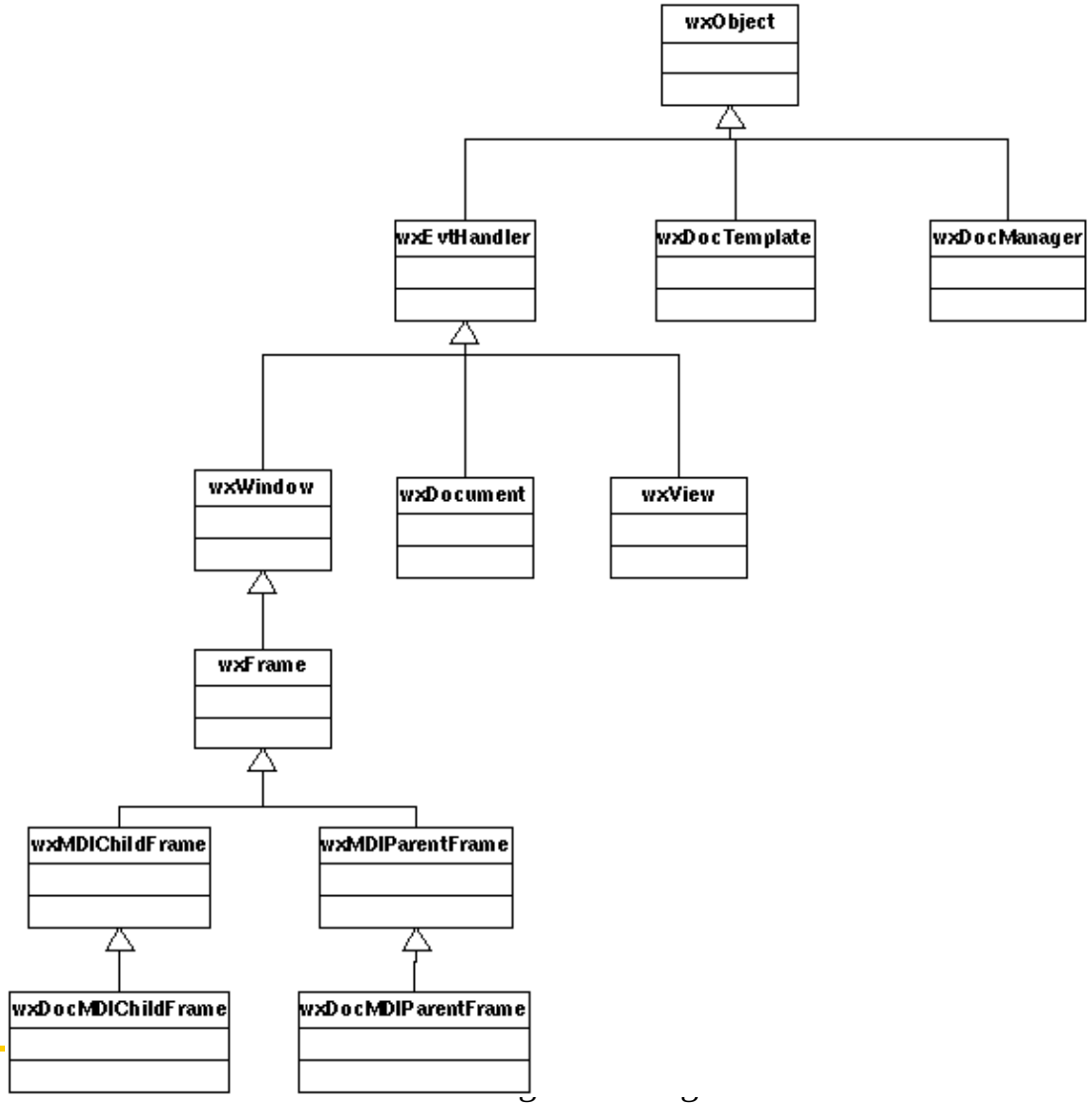
- `": public CVehicle"` genomför ärvningen, så att alla egenskaper hos CVehicle överförs till den nya klassen

Ärvning

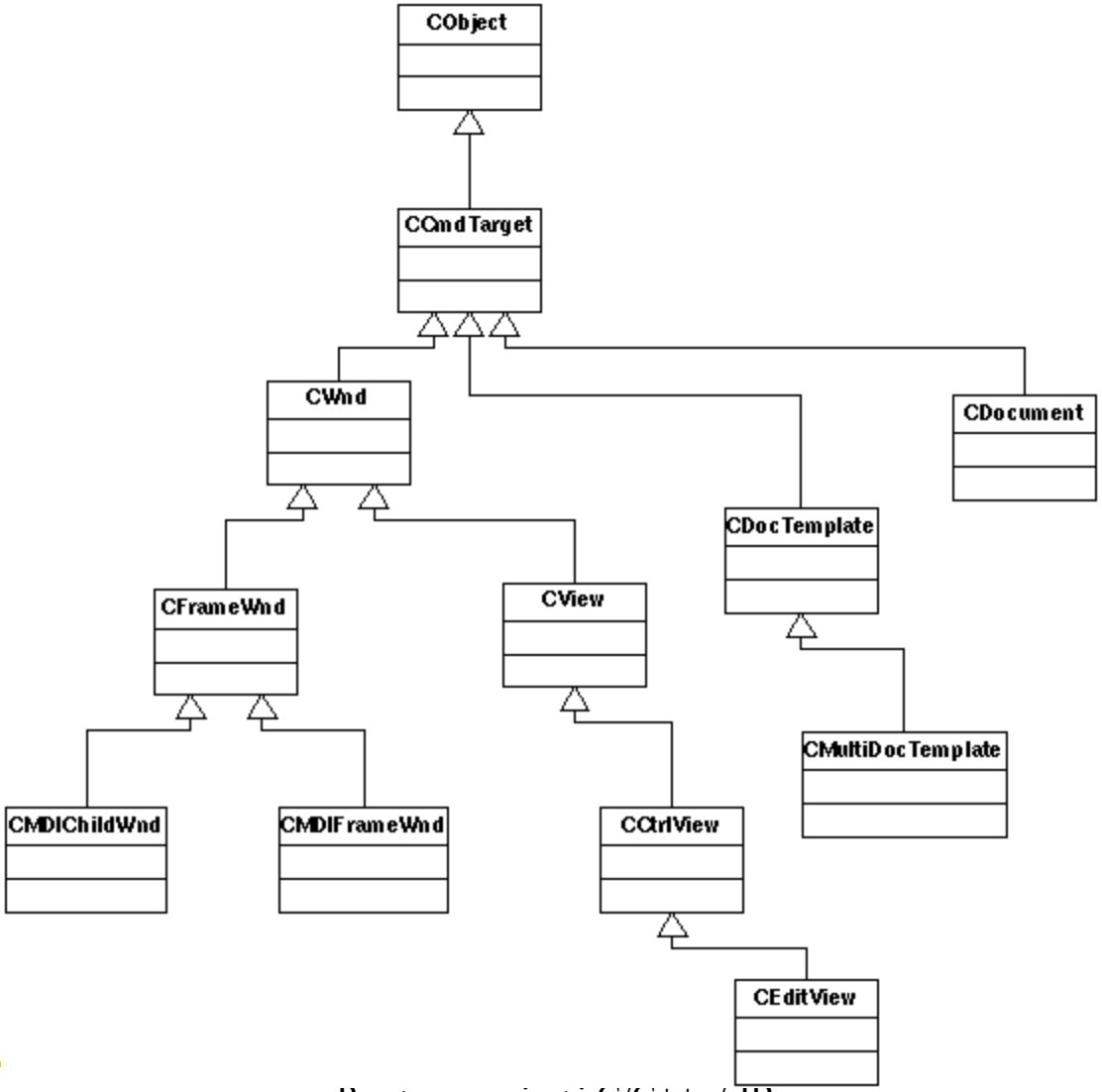


- **Dvs en CCar är en speciell typ av CVehicle**
- **Märk:**
  - **Privata (private:) klassmedlemmar är fortfarande privata, den ärvande klassen måste accessera dessa via funktioner**

wxWidgets klasshierarki



MFC klasshierarki



Konstruktorer för ärvda klasser

- Version ”dålig”

```
CCar::CCar() (float weight, unsigned speed) {
   SetWeight(weight);
   SetSpeed(speed);
}
```

- ”Dålig”, eftersom
  - 1. Default construktor för CVehicle anropas, varvid klassmedlemmarnas initialiseras
  - 2. CCar:s konstruktor kallas, varvid klassmedlemmarna IGEN sätts till nytt värde
- Detta borde ske samtidigt, så att endast en konstruktor kallas

Konstruktor för ärvda klasser

- Version ”OK”

```
CCar::CCar() (float weight, unsigned speed)
: CVehicle(weight){
   SetSpeed(speed);
}
```

- Istället kallas basklassens konstruktor nu direkt med motsvarande argument från deriverade klassen

Destruktorer för ärvda klasser

- Då ett objekt frigörs, kommer samtliga destruktorer för de sammansatta klasserna att exekveras

```
class Base {
   public:
    ~Base();
};
class Derived: public Base {
   public:
    ~Derived();
};
int main() {
   Derived derived;
}
```

I vilken ordning kallas konstruktorer / destruktorer ??

```
// constr.cpp

class Base {
public:
   Base();
   ~Base();
};

class Derived:public Base {
public:
   Derived();
   ~Derived();
};

class Derived2:public Derived
   {
public:
   Derived2();
   ~Derived2();
};
```

```
Base::Base() {printf("Base class
   constructor\n"); }
Base::~Base() {printf("Base class
   destructor\n"); }


Derived::Derived() {printf("Dervied
   class constructor\n"); }
Derived::~Derived() {printf("Derived
   class destructor\n"); }


Derived2::Derived2() {printf("Dervied2
   class constructor\n"); }
Derived2::~Derived2() {printf("Derived2
   class destructor\n"); }



int main(int argc, char* argv[])
{
   printf("Main starting\n");
   Derived2 der;
   printf("Main ending\n");

   return 0;
}
```

I vilken ordning kallas konstruktorer / destruktorer ??

```
% g++ constr.cpp –o constr
% ./constr

Main starting
Base class construktor
Dervied class construktor
Dervied2 class construktor
Main ending
Derived2 class destruktor
Derived class destruktor
Base class destruktor
```

Omdefiniering av medlemsfunktioner

- Funktioner som ingår i basklasser kan bli omdefinierade
  - T.ex. vikt för bil räknas genom att till basvikten räkna till en ”beräknad förarvikt”

```
float CCar::GetWeight() {
   float tmpWeight = CVehicle::GetWeight()+80.0;
   return tmpWeight;
}
```

- Om man fortfarande vill komma åt CVehicles gömda funktion, går det genom

```
CVehice::GetWeight(); /* eller för instantierade klass */
CCar car;
w = car.CVehicle::GetWeight();
```

Multipel ärvning

- Ärvningen kan också ske från flera klasser, varvid man talar om *multiple inheritance*

```
class CCar: public CVehice, public CEngine {
public:
   CCar();
   CCar(float w, unsigned speed, unsigned kw);
}

CCar::CCar(float w, unsgined speed, unsigned kw):
   CVehicle(w), CEngine(kw) {
   SetSpeed(speed);
}
```

- Istället kallas basklassens konstruktor nu direkt med motsvarande argument från deriverade klassen

Konversion mellan basklass och deriverad klass

- En deriverad klass kan automatiskt också ses som en basklass

```
CCar car(800.0, 180);
CVehicle veh(40.0);
veh = car;      /* OK veh subset av car */
car = veh;      /* Inte OK, hur t.ex avgöra värden för
   variabler som saknas i CVehicle ? */
CVehicle *vp = &car; /* OK, men endast CVehicles
           funktionalitet kan användas */
CCar *cp = &veh;   /* Inte OK, vi kan inte använda en
           pekare till deriverad klass till
           att peka på basklass */
float w=vp->GetWeight(); /* Detta ger dock CVehicle-klassens
       version av fordonstyngd (ej +80kg...) */
```

Polymorfism

- Normalt gäller att vi använder funktion enligt pekartyp
  - Men, C++ gör det möjligt att använda den deriverade klassens funktion trots att man använder sig av en pekare till basklassen → polymorfism
    - dvs. vp->GetWeight() ger 880 istället för 800 i exemplet på förra sidan
  - Genomförs via ”late-binding”, dvs länkning under exekvering, först då avgörs vilken funktion som verkligen skall anropas

Polymorfism

- Polymorfism, genomförs via direktivet virtual för funktionsnamnet. Efter detta kommer den deriverade klassens motsvarande funktion att användas, även om en pekare till basklassen används.

```
class CVehicle {
public:
   CVehicle();
   CVehicle(float w);
   virtual float GetWeight();
private:
   float m_Weight;
}
```

- Notera, efter att en funktion i basklass har deklarerats *virtual*, kommer den att förbli det i alla deriverade klasser.

Rent virtuella funktioner

(Pure virtual funtions)

- Hittills har vi antagit att den virtuella funktionen också har en implementation i basklassen. Om vi inte har någon implementation i basklassen, talar man om en *rent virtuell funktion* (*pure virtual functions)*.
  - En rent virtuell funktion skapas genom att till funktionsdeklarationen addera prefixet virtual och postfixet =0

```
class CVehicle {
    virtual float GetWeight() = 0;
    virtual float GetSpeed() = 0;
}
```

- Vi har deklarerat ett *protokoll*, m.a.o. regler för vad som måste implementeras innan en deriverad klass kan användas

Dynamiskt minnesallokering i C++

- Hittills har vi behandlat statiska eller automatiska klasser
- C++ inför *operatorerna* **`new`** och **`delete`** för att allokera dynamisk minne

```
main() {
 CVehicle *vp = new CVehicle(12.0);
     ... // Code
 delete vp; /* kallar även på destruktorn */
}
```

- Notera att **`new`** och **`delete`** också kallar på konstruktorn/destruktorn

Dynamiskt minnesallokering i C++

- Allokering av räckor
  - **`CVehicle *vp = new CVehicle[40];`**
  - Explicit allokering
  - Vid allokering av räckor kallas alltid default constructor (konstruktor utan argument)
- Deallokering av räckor
  - Deallokering genom

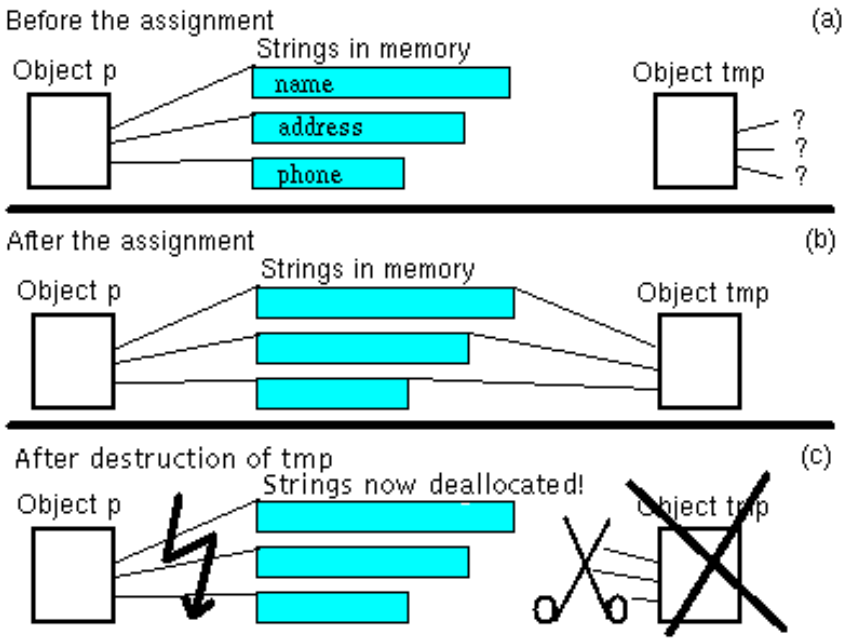
```
delete [] vp; /* Destruktorerna kallas för
                 varje objekt skillt */
delete vp;    /* Destruktor endast för
                 första dymaniska fältet */
```

Tilldelnings-operatorn (=)

- I C++ kan vi tilldela värden åt strukturer och klasser precis som vi tilldelar åt heltal etc.

–I praktiken utförs implicit kopiering bit för bit från en struktur till den andra

–Om strukturer innehåller pekare blir et problem, ty →



Copy-constructorn

```
class Person {
  public:
   Person(Person const &other) {Copy(other);}
   Copy(Person const &other);
  }
void Person::Copy(Person const &other) {
   delete m_namn;
   delete m_adress;

   m_namn = strdupnew(other.m_namn);
   m_adress = strdupnew(other.m_adress);
  }
```

Överladdning av operatorer

- I C++ kan vi tilldela operatorer ny funktionalitet
  - t.ex. för assignment =

```
class Person {
public:
 void operator=(Person const &other);
}
void Person::operator=(Person const &other) {
 delete m_namn;
 delete m_adress;

 m_namn = strdupnew(other.m_namn);
 m_adress = strdupnew(other.m_adress);
}
```

Överladdning av operatorer

- Vi kan överladda de fleste operatorer, t.ex.

```
class Person {
    char *m_namn;
    char *m_adress;
public:
    int operator>(const Person &other);
    Person(char *, char*);
    Person() {m_namn=m_adress=0;}
};
Person::Person(char *n, char *a) {
    m_namn = strdupnew(n);
    m_adress = strdupnew(a);
}

int Person::operator>(const Person &other) {
    return(strcmp(m_namn, other.m_namn)>0?1:0);
}
main() {
    Person a("axel", "borg"), b("Test", "Addr");
    if (a>b) printf ("a>b\n");
}
```

Överladdning av []-operatorn -exempel

```
class IntArray {
   int *d_data;
   unsigned d_size;
public:
   IntArray(unsigned size = 1);
   IntArray(IntArray const &other);
   ~IntArray();
   IntArray const &operator=(IntArray const &other); // overloaded
   index operators:
   int &operator[](unsigned index); // first
   int const &operator[](unsigned index) const; // second
private:
   void boundary(unsigned index) const;
   void copy(IntArray const &other);
   int &operatorIndex(unsigned index) const;
};
```

Överladdning av []-operatorn -exempel

```
IntArray::IntArray(unsigned size)
    : d_size(size) {
        if (d_size < 1) {
            cerr << "IntArray: size of array must be >= 1\n";
            exit(1);
        }
        d_data = new int [d_size];
    }
 IntArray::IntArray(IntArray const &other) {
        copy(other);
    }

    IntArray::~IntArray() {
        delete d_data;
    }

    IntArray const &IntArray::operator=(IntArray const &other) {
        if (this != &other) {
            delete d_data;
            copy(other);
        }
        return *this;
    }
```

Överladdning av []-operatorn -exempel

```
void IntArray::copy(IntArray const &other) {
        d_size = other.d_size;
        d_data = new int [d_size];
        memcpy(d_data, other.d_data, d_size * sizeof(int));
    }

    int &IntArray::operatorIndex(unsigned index) const {
        boundary(index);
        return d_data[index];
    }

    int &IntArray::operator[](unsigned index) {
        return operatorIndex(index);
    }

    int const &IntArray::operator[](unsigned index) const {
        return operatorIndex(index);
    }

    void IntArray::boundary(unsigned index) const {
        if (index >= d_size) {
            cerr << "IntArray: boundary overflow, index = " <<
                    index << ", should range from 0 to " << d_size - 1 << endl;
            exit(1);
        }
    }
```

Exempel: Komplexa tal

```
class Complex {

public:
          // Default constructor + generell constructor
    Complex(double re=0.0, double img=0.0) {m_re=re;m_img=img;}
    Complex(const Complex &other) {Copy(other);}

    Complex &operator=(const Complex &other) {Copy(other); return *this;}
    Complex operator*(const Complex &other) const;
    Complex operator+(const Complex &other) const;
    Complex operator/(const Complex &other) const;

    double Re() {return m_re;}
    double Img() {return m_img;}
    const char *c_str(); // Formattera som text-sträng
private:
    void Copy(const Complex &other) {m_re=other.m_re; m_img=other.m_img;}
    double m_re;
    double m_img;
    char m_c_str[20];
};
```

Exempel: Komplexa tal 2

```
// (ac-bd) + i(ad+bc)
Complex Complex::operator*(const Complex &other) const {
    Complex res;
    res.m_re = m_re*other.m_re - m_img*other.m_img;
    res.m_img = m_re*other.m_img + m_img*other.m_re;
    return res;
}

//  (ac+bd) + i(bc-ad)
//  ------------------
//      c*c+d*d
Complex Complex::operator/(const Complex &other) const {
    Complex res;
    double den = other.m_re*other.m_re + other.m_img*other.m_img;
    res.m_re = (m_re*other.m_re + m_img*other.m_img) / den;
    res.m_img = (m_img*other.m_re - m_re*other.m_img) / den;
    return res;
}

const char *Complex::c_str() {
    sprintf(m_c_str, "%f%s%fj", m_re, m_img<0?"":"+", m_img);
    return m_c_str;
}
```

Exempel: Komplexa tal 3

```
int main(int argc, char* argv[])
{
   Complex b(3.4, 1.2);  // Tal 1
   Complex c(2.2, -5.2); // Tal 2
   Complex sum = b+c;    // Summan
   Complex prod;     // Produkt
   prod = b*c;       // Beräkna produkt

   printf ("sum of %s and %s is %s\n", b.c_str(), c.c_str(),
   sum.c_str());
   printf ("prod of %s and %s is %s\n", b.c_str(), c.c_str(),
   prod.c_str());
   printf ("div of %s and %s is %s\n", b.c_str(), c.c_str(),
   (b/c).c_str());

   return 0;
}
```

”this”-pekaren

- Då man använder sig av methoder för objekt, har man tillgång till pekaren ”this”
  - Pekar alltid på det nuvarande objektet, oberoende om det är på stacken, heapen eller datasegmentet

```
//T.ex. undvik kopiering av sig själv
MyClass &Copy(MyClass const &other) {
  if (this != &other) {
    ... // Gör endast om ej självtilldelning
  }
  return *this;
}
```

Operatorer som kan överladdas

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

Exceptioner

- Vid felsituationer i C, sker ofta något av följand
  - Funktionen noterar det ovanliga, och ger ett meddelande
  - Funktionen stoppar exekveringen och returnerar ett felmeddelande till kallaren
    - Den kallande funktionen kan i sin tur föra felmeddelanden uppåt i hierarkin
  - Funktionen gör beslutet att saker spårar ur och avslutar processen med t.ex. `exit()`
  - Funktionen använder en kombination av `setjmp()` och `longjmp()`, för att direkt återvända till en högre nivå

Exceptioner i C++

- C++ erbjuder naturligtvis alla föregående sätt att hantera felsituationer
- C++ erbjuder dessutom exceptioner, eller sätt att kontrollerat återvända i call-stacken till en högre nivå
- När funktionen själv inte kan hantera felet på ett vettigt sätt, är exceptioner ett gott alternativ

Exceptioner: Syntax

```
try {
  // kod, inom vilken exceptioner kan
  // genereras
} catch (<type> <var>) {
  // Vad som händer, om en exception av
  // given typ genereras
}

// Exceptioner genereras genom
throw ”This generates a char * exception;
```

Exceptioner: hierarki

- try-block kan finnas på olika nivåer, t.ex.
  - main(): huvudnivå, hanterar exceptioner om inte andra hanterar hittas
  - i funktioner: beroende på funktion, kan denna hantera exceptionerna

```
int main() {
 try {
 myfunc1();
 myfunc2();
 } catch (...) {
     printf(”Exception\n”);
}
void myfunc1() {
 try {
     // Egen hanterare
 } catch(int i) { //kod }
}
```

Exceptioner: default catcher

```
try {
   // Kod
} catch (...) { //Default catcher
   printf(”Nåt fel uppstod (vad sägs om out of
   memory??\n”);
}

try {
   try {
    throw 12.33;
   } catch (char *s) {
    printf (”Char * exception\n”);
   } catch (...) {
    printf(”Generic error handler\n”);
    throw;
   }
} catch (double d) {
   printf(”Outer level double exception catcher\n”);
}
```

Exceptioner som inte fångas

- En exception som inte ”fångas” går till
  - **`std::terminate()`**
- `std::terminate()` kallas i sin tur på
  - `std::abort()`

Exceptioner i konstruktorer

- Om man genererar en exception i en konstruktor så att exceptionen sker inne i konstruktorn, kommer destruktorn inte att kallas
  - Om man allokerat minne före denna ”förlorade” exception, kommer en minnesläcka att uppstå
- Regel: En exception som genereras i en konstruktor skall inte okontrollerat gå vidare upp i hierarkin

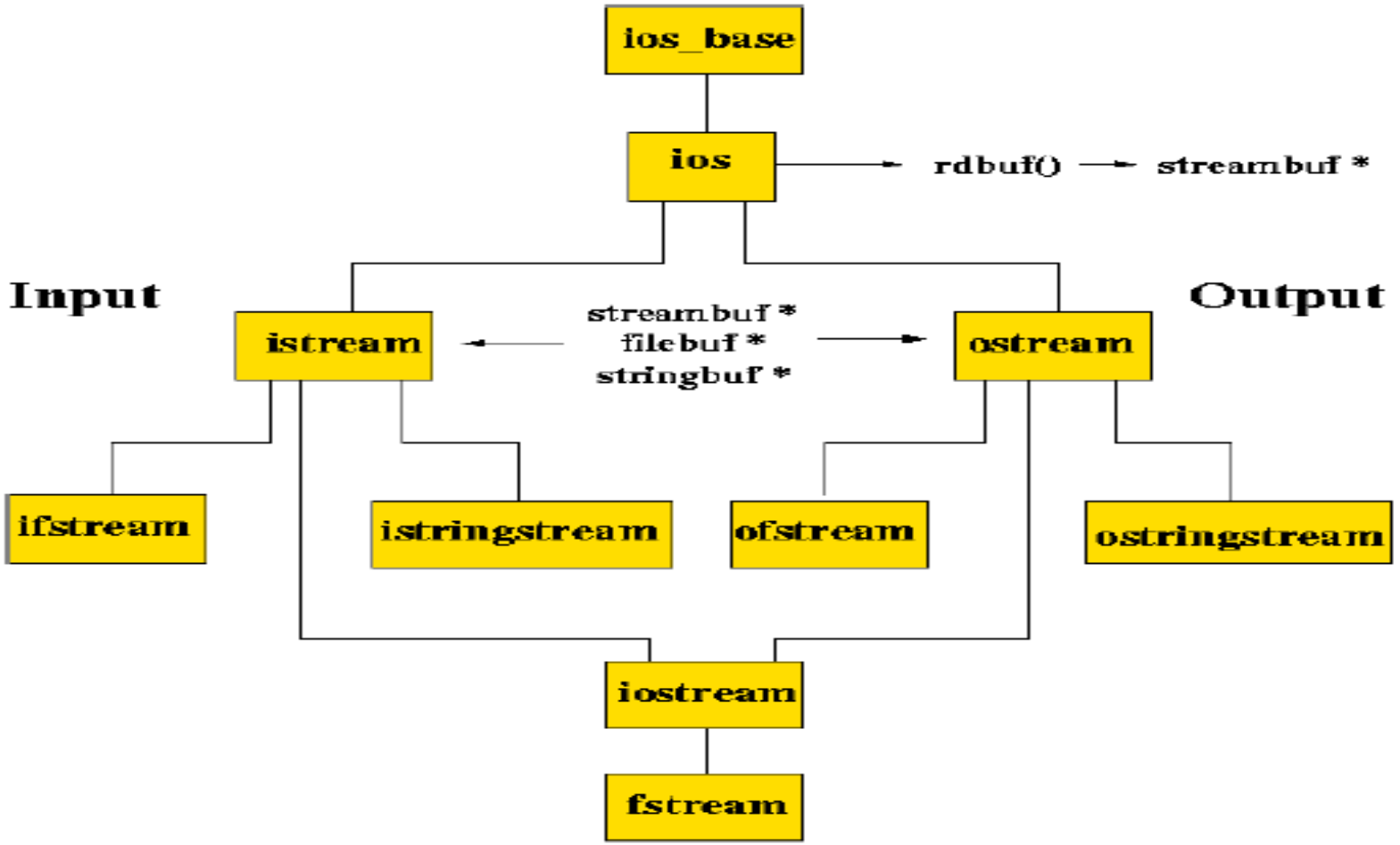
iostream-biblioteket

- I C++ definieras i/o-strömmarna **`cout, cin, cerr`** genom att inkludera filen `<iostream>`

```
#include <iostream>
main() {
 int nval;
 cout << ”Ge ett nummer” <<endl;
 cin >> nval;
 cout << ”Numret du gav var: ”<< nval << endl;
}
```

- **`cout,cin, cerr`** är de fakto objekt-instanser av motsvarande klasser, ”<<” och ”>>” är överladdade operatorer

Centrala i/o-klasser



iostream....

- Genom att använda överladdade operatorer kan man ”mata in” olika variabler till stream-biblioteket

```
#include <iostream>
  main() {
   int nval;
   float f;
   double d;
   cout << ”Ge ett nummer” <<endl;
   cin >> nval;
   f = nval; d = nval;
   cout << ”Numret du gav var: ”<< nval << endl;
   cout << Flyttal: ”<< f << endl;
   cout << Dubbel prec.: ”<< d << endl;
  }
```

iostream

- Ger säkerhet till i/o, jfr C: printf

```
double bt = (double)133/(double)5;
printf(”Bilar per timme %i\n”, bt);
```

  – ger svaret –1717986918, p.g.a. fel i i matchingen format <-> parametrar

```
cout << ”Bilar per timme ” <<
  (double)133/(double)5<<endl;
```

  ger däremot rätt svar 26.6

iostream-formattering -utdrag

- `ios::adjustfield`:
  - mask value used in combination with a flag setting defining the way values are adjusted in wide fields (`ios::left`, `ios::right`, `ios::internal`). Example, setting the value 10 left-aligned in a field of 10 character positions:

```
cout.setf(ios::left, ios::adjustfield);
cout << "'" << setw(10) << 10 << "'" << endl;
```

- `ios::basefield`:
  - mask value used in combination with a flag setting the radix of integral values to output (`ios::dec`, `ios::hex` or `ios::oct`). Example, printing the value 57005 as a hexadecimal number:

```
cout.setf(ios::hex, ios::basefield);
cout << 57005 << endl;
// or, using the manipulator:
cout << hex << 57005 << endl
```

OBS! Mera i t.ex. C++-annotations

Utskrift till fil

```
#include <iostream>
int main()
{
ofstream of;
cout << "of’s open state: " << boolalpha <<
  of.is_open() << endl;
of.open("/tmp/myfile"); // on Unix systems
cout << "of’s open state: " << of.is_open() <<
  endl;
of << ”Testing output to file” << (float)5.44;

}
```

Generisk programmering

- C++ erbjuder möjligheter att definiera och implementera generella (eller abstrakta) funktioner och klasser
- Detta system kallas *Template Mechanism*
- Genom att specificera ett template, kan kompilatorn generera kod, som är anpassad till den egentliga datatyp som används
- Template-mekanismen kan ses som jobbig i början, men efterhand brukar man se nytta av den

Generisk programmering: intro

```
// Generell funktion: Addera två argument
Type add(Type const &lvalue, Type const &rvalue) {
  return lvalue + rvalue;
}
//För type double
double add(double const &lvalue, double const
  &rvalue) {
  return lvalue + rvalue;
}
```

– Detta kan förstås upprepas förvarje tänkbar datatyp, användande av överladdade funktioner... men leder till ...många... versioner av funktionen

Generisk funktion

```
template <typename Type>
Type add(Type const &lvalue, Type const
  &rvalue) {
  return lvalue + rvalue;
}

// Nu kan vi använda
double a=1.2, b=3.34;
printf(”sum of %f and %f is %f\n”, a, b,
  add(a,b));

// Kompilatorn genererar nu en överladdad
  funktion av rätt typ
```

Format på template-funktion

Keyword

Kommaseparerad lista med templat-parameterlista

```
template <typename Type>
Type add(Type const &lvalue, Type const
  &rvalue) {
  return lvalue + rvalue;
}
```

Vi kan notera, att hittills har vi endast använt formella variabelnamn som argument till funktioner, typerna har varit givna.

Nu låter vi även typerna på argumenten fungera som formella namn.

Exempel med den komplexa datatypen

```
template <typename Type>
Type add(Type const &lvalue, Type const
  &rvalue) {
  return lvalue + rvalue;
}

int main() {
  cout << "Sum of " << x << " and " << y <<" =
  " << add(b,c) <<endl;
}
```

Hur avgörs argumenttyp??

- Endast funktionargument används, inte lokala variabler eller retur-värden
- 3(4) konversioner kan användas
  - lvalue till rvalue
  - tilläggande av const till icke-const
  - konversion till bas-klass
  - standard konversioner (int to double, int to unsigned etc)

Template-klasser

- Används i STL (Standard Template Library)
- Typisk för klasser som enkapsulerar andra datatyper såsom
  - Vektorer
  - Matriser
  - Länkade listor

Komplexa klassen med template

```
template <typename T1>
class Complex {
 public:
  Complex(T1 re=0, T1 img=0) {m_re=re;m_img=img;}
  Complex(const Complex &other) {Copy(other);}
  Complex<T1> &operator=(const Complex<T1> &other) {Copy(other); return
   *this;}
  Complex<T1> operator*(const Complex<T1> &other) const;
  Complex<T1> operator/(const Complex<T1> &other) const;
  Complex<T1> operator+ (const Complex<T1> &other) const;
  //Complex const operator+(const Complex &other) {Complex res(*this);
   res.m_re+=other.m_re;res.m_img+=other.m_img; return res;\
}
  int a;
  //ostream &operator<<(ostream &stream, Complex const &c);

  T1 const Re() const {return m_re;}
  T1 const Img() const {return m_img;}
  const char *c_str();
 private:
  void Copy(const Complex &other) {m_re=other.m_re; m_img=other.m_img;}
  T1 m_re;
  T1 m_img;
  char m_c_str[20];
};

template <typename T1>
ostream &operator<< (ostream &, Complex<T1> const &);
```

Komplexa tal 2

```
/// (ac-bd) + i(ad+bc)?
template <typename T1>
Complex<T1> Complex<T1>::operator*(const Complex<T1> &other) const {
  Complex res;
  res.m_re = m_re*other.m_re - m_img*other.m_img;
  res.m_img = m_re*other.m_img + m_img*other.m_re;
  return res;
}
//  (ac+bd) + i(bc-ad)?
//  ------------------
//      c*c+d*d
template <typename T1>
Complex<T1> Complex<T1>::operator/(const Complex<T1> &other) const{
  Complex<T1> res;
  double den = other.m_re*other.m_re + other.m_img*other.m_img;
  res.m_re = (m_re*other.m_re + m_img*other.m_img) / den;
  res.m_img = (m_img*other.m_re - m_re*other.m_img) / den;
  return res;
}
template <typename T1>
ostream &operator<<(ostream &os, Complex<T1> const &c) {
  return os <<
    "(" << c.Re() << "," <<
    c.Img() << "j)";
}
```

Komplexa 3

```
//typedef Complex<double> complex; // Komplex klass med double som element
typedef Complex<int> complex; // Komplexx klass med heltal som element

int main(int argc, char* argv[])
{
  complex b(3, 1);
  complex c(2, -5);
  complex sum = b+c;
  complex prod;
  prod = b*c;

  using namespace std;

  printf ("sum of %s and %s is %s\n", b.c_str(), c.c_str(), sum.c_str());
  printf ("prod of %s and %s is %s\n", b.c_str(), c.c_str(), prod.c_str());
  printf ("div of %s and %s is %s\n", b.c_str(), c.c_str(), (b/c).c_str());

  cout << "Sum of " << b << " and " << c <<" = " << (b+c) <<endl;
}
```

Ex: Stack

```
template <class T>
class Stack {
public:
Stack(int = 10) ;
~Stack() { delete [] stackPtr ; }
int push(const T&);
int pop(T&) ;
int isEmpty()const { return top == -1 ; }
int isFull() const { return top == size - 1 ; }
private:
int size ; // number of elements on Stack.
int top ;
T* stackPtr ; } ;
```

Ex: Stack (2)

```
//constructor with the default size 10
template <class T>
Stack<T>::Stack(int s) {
  size = s > 0 && s < 1000 ? s : 10 ;
  top = -1 ; // initialize stack
  stackPtr = new T[size] ;
}
// push an element onto the Stack
template <class T>
int Stack<T>::push(const T& item) {
   if (!isFull()) {
    stackPtr[++top] = item ;
    return 1 ; // push successful
    }
  return 0 ; // push unsuccessful
}
```

Ex: Stack (3)

```
// pop an element off the Stack
template <class T>
int Stack<T>::pop(T& popValue) {
  if (!isEmpty()) {
     popValue = stackPtr[top--] ;
     return 1 ; // pop successful
  }
  return 0 ; // pop unsuccessful
}
```

Ex: Stack (4)

```
#include <iostream>
#include "stack.h"
using namespace std ;
void main() {
  typedef Stack<float> FloatStack ;
  typedef Stack<int> IntStack ;
  FloatStack fs(5) ;
  float f = 1.1 ;
   cout << "Pushing elements onto fs" << endl ;
   while (fs.push(f)) { cout << f << ' ' ; f += 1.1 ; }
   cout << endl << "Stack Full." << endl << endl << "Popping elements
   from fs" << endl ;
   while (fs.pop(f))
     cout << f << ' ' ; cout << endl << "Stack Empty" << endl ;
   cout << endl ;
   IntStack is ;
   int i = 1.1 ;
   cout << "Pushing elements onto is" << endl ;
    while (is.push(i)) { cout << i << ' ' ; i += 1 ; }
    cout << endl << "Stack Full" << endl << endl << "Popping elements
   from is" << endl ;
   while (is.pop(i)) cout << i << ' ' ; cout << endl << "Stack Empty"
   << endl ;
}
```