

## Härledda klasser

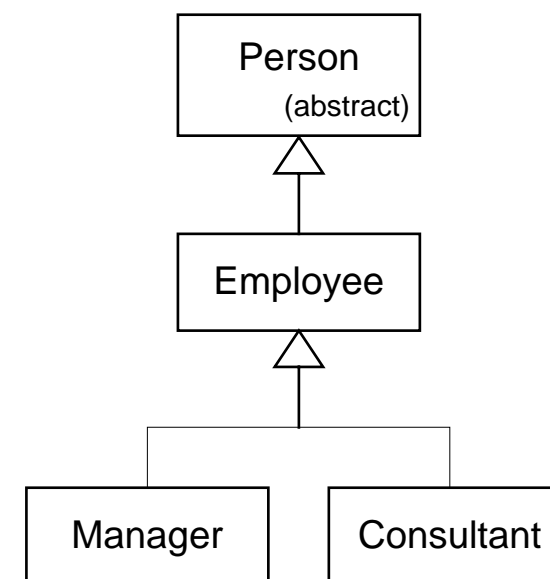
C++ har en relativt komplett och därmed komplicerad modell för härledning/ arv

- stödjer flera sätt för subclasser att ärva från sina basklasser
  - *enkelt* arv – endast en direkt basklass
  - *multipelt* arv – två eller flera direkta basklasser
  - *upprepat* arv – en indirekt basklass ärvs flera gånger via multipelt arv
  - multipelt och upprepat arv kan leda till tvetydigheter och andra problem – behov av mekanismer för att lösa detta
- flera sätt att specificera *tillgänglighet för basklassmedlemmar* i den härledda klass
  - **public** basklass – **public**-medlemmar i basklassen blir **public** i den härledda klassen (**protected** blir **protected**)
  - **protected** basklass – **public**-medlemmar i basklassen blir **protected** i den härledda klassen (**protected** blir **protected**)
  - **private** basklass – **public**-medlemmar i basklassen blir **private** i den härledda klassen (**protected** blir **private**)
  - tillgängligheten är **public** om basen är en **struct**, **private** om en **class** ifall inget anges
  - en klass kan utse *vänner* – en **friend** har åtkomst till alla medlemmar, även privata
- polymorft beteende bestäms av programmeraren
  - objekt som ska bete sig polymorft måste refereras via pekare eller referenser
  - endast anrop av *virtuella* medlemsfunktioner kan bindas dynamiskt och därmed uppvisa polymorft beteende
  - en icke-polymorf klasshierarki använder arv för återanvändning av kod för, i princip, fristående klasser

## Person-Employee-Manager-Consultant – en polymorf klasshierarki

Konstruktion av en enkel *polymorf klasshierarki* för att hantera anställda (*employees*) vid ett företag.

- en klass för att representera *personer* i allmänhet – **Person**
  - har namn och personnummer (*civic registration number*)
  - alla anställda ska dela egenskaperna hos denna klass
  - inga objekt ska kunna skapas – ska vara en *abstrakt klass*
- en klass för anställda i allmänhet – **Employee**
  - har anställningsdatum, anställningsnummer, lön, anställning vid en avdelning
  - mer specialiserade kategorier av anställda ska härledas från denna klass
  - objekt ska kunna skapas – ska vara en *konkret klass*
- en klass för anställda som är avdelningschefer – **Manager**
  - ansvarar för en avdelning och dess anställda
- en klass för (tillfälligt) anställda som är konsulter – **Consultant**
  - ingen direkt skillnad jämfört med en anställd i allmänhet men ska vara särskiljbar typmässigt
- objekt skapas typiskt dynamiskt och hanteras med pekare
  - annars inget polymorft beteende hos objekt



**Observera**, I det fullständiga kodexemplet har medlemsfunktioner som implementeras med endast en rad **definierats** i klassdefinitionerna – i exemplen som visas på följande sidor **deklarerar** endast funktionerna av utrymmesskäl.

## Class Person

```

class Person
{
public:
    virtual ~Person() = default; // HIC++ 12.5.2

    virtual std::string str() const;
    virtual Person* clone() const = 0;

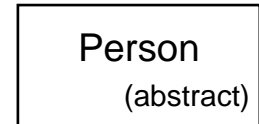
    std::string get_name() const;
    void set_name(const std::string&);
    CRN get_crn() const;
    void set_crn(const CRN&);

protected:
    Person(const std::string& name, const CRN& crn);
    Person(const Person&) = default; // HIC++ 12.5.2

private:
    Person& operator=(const Person&) = delete; // HIC++ 12.5.8

    std::string name_;
    CRN crn_;
};

```



- defaultkonstruktor, flyttkonstruktor och flytttilldelningsoperator genereras *inte*
- alla speciella medlemsfunktioner som annars skulle ha genererats implicit är deklarerade (kodningsregel)

## Kommentarer till Person

- i grunden en trivial klass
  - välartade datamedlemmar – båda har defaultkonstruktion och destruering
  - de kompilatorgenererade speciella medlemsfunktionerna är i grunden bra men tillåts endast för internt bruk
- ”*defaulted*” (= **default**) och ”*deleted*” (= **delete**) medlemsfunktioner
  - *defaultkonstruktor* i genereras inte om någon annan konstruktor deklarerar – kan ”*defaultas*” om önskad
  - *kopieringstilldelning* ska inte vara tillåten – åtkomstspecifikationen är betydelselös men underförstått **private** om ”*deleted*”
  - *flyttilldelning* genereras *inte* om destruktor, kopieringskonstruktor eller kopieringstilldelningoperator deklarerar
  - *regel*: deklarerar alltid speciella medlemsfunktioner som annars hade genererats för att tydligt dokumentera gränssnittet
- *virtuella funktioner (virtual)*
  - virtuella funktioner kan *överskuggas* av subklasser – deklarerar med *override* (*HIC++ 10.2.1*)
  - sker om en funktion med samma signatur deklarerar i en subklass – deklarerar inte **virtual** i subklasser
  - gör klassen *polymorf*
- *pure virtual funktion*
  - *pure specifier* = 0 gör att en virtuell funktion inte kan anropas en publik medlem
  - *kan* ha en separat definition – *måste* om en destruktor – anropbar från andra medlemsfunktioner och subklassers medlemsfunktioner
  - pure virtual funktioner ärvs – en subklass blir också abstrakt såvida inte alla ärvda pure virtual funktioner överskuggas
  - gör klassen *abstrakt*

## Kommentarer till Person, forts.

- *polymorf klass*
  - har virtuella funktioner, egna eller ärvda
  - måste ha en *virtuell destruktör* för att säkerställa korrekt destruering av subobjekt (*HIC++ 12.2.1*)
  - objekt kommer att innehålla *typinformation* – används vid anrop av virtuella funktioner och av **dynamic\_cast**
  - objekts kommer att innehålla en *virtuell tabell* (t.ex. `__vtable`) – implementeringsteknik för anrop av virtuella funktions (skapas av kompilatorn)
- *abstrakt klass*
  - inga fristående Person-objekt av kan skapas
- *skyddade konstruktörer*
  - eftersom Person är abstrakt finns det inget behov av publika konstruktörer
  - **protected** konstruktörer används för att framhäva abstraktheten – inget annat syfte finns
- *statisk typ* och *dynamisk typ*

```
Person* p = new Employee(...); // p har statisk typ "pekare till Person"
```

```
p->clone(); // den dynamiska typen för uttrycket *p är Employee
```

- *statisk typ* används under kompilering för att kontrollera att `clone()` är tillåten för den typ av objekt `p` kan peka på
- *dynamisk typ* används vid exekvering för att binda den överskuggning av `clone()` som gäller för det objekt som `p` pekar på

## *Konstruktor som tar namn och personnummer*

```
Person::Person(const std::string& name, const CRN& crn)
    : name_{ name }, crn_{ crn }
    {}
```

Säkerställer att en ny Person alltid har ett namn och ett personnummer

- defaultkonstruktorn genereras inte
- ingen annan konstruktor är tillgänglig som kan initiera objekt på något annat sätt, utom kopieringskonstruktorn och flyttkonstruktorn
- ska endast användas av motsvarande konstruktor i subklasserna – **protected**

## Medlemsfunktionen `str()`

```
virtual std::string str() const;
```

Definition:

```
string Person::str() const
{
    return name_ + ' ' + crn_.str();
}
```

Anrop kommer att bindas *dynamiskt*, om objektet refereras av pekare eller referens.

- den dynamiska typen avgör vilken överskuggning som anropas

```
Person* p{ new Manager{name, crn, date, employment_number, salary, dept} };
```

```
cout << p->str() << endl;
```

- pekaren `p` har *statisk typ* `Person*`
- uttrycket `*p` har *dynamisk typ* `Manager`

```
(*p).str()
```

- `Manager::str()` anropas – vi föredrar piloperatorn i detta fall

```
p->str()
```

## Medlemsfunktionen `clone()`

```
virtual Person* clone() const = 0;
```

Polymorfa klasser behöver ibland en *polymorf kopieringsfunktion*.

- använder man polymorfa klasser innebär det ibland att man ska allokeras objekten dynamiskt och hanterar dem via pekare
  - kräver en polymorf kopieringsfunktion som `clone()`
  - varje konkret subclass måste ha sin egen, specifika överskuggning av `clone()`
- lämplig kandidat för att göra `Person` *abstrakt*
  - inga fristående `Person`-objekt ska kunna skapas
  - deklarerar *pure virtual*, ”= 0” (*pure specifier*)
  - ingen definition ska (kan) finnas i detta fall
- standardbibliotekets strömklasser är exempel på polymorfa klasser som *inte* ska kunna kopieras
  - saknar publik kopieringskonstruktor



## Subklassen Employee

```

class Employee : public Person
{
public:
    Employee(const std::string& name,
             const CRN& crn,
             const Date& e_date,
             int e_number,
             double salary,
             int dept = 0);

    ~Employee() = default;

    std::string str() const override;
    Employee* clone() const override;

    int get_department() const;
    Date get_employment_date() const;
    int get_employment_number() const;
    double get_salary() const;

protected:
    Employee(const Employee&) = default;

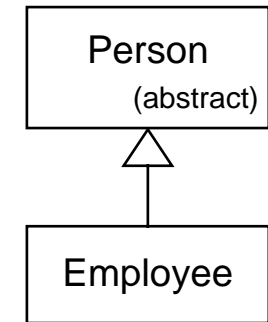
```

// *HIC++ 12.5.1, 12.5.2*

// *HIC++ 10.2.1*

// *observera returtypen – kovariant med Person\**

// *HIC++ 12.5.1, 12.5.2*



```
private:
    Employee& operator=(const Employee&) = delete; // HIC++ 12.5.1

    friend class Manager; // HIC++ 11.2.1 (använd inte friend)
    void set_department(int dept);
    void set_salary(double salary);

    IDA_date::Date e_date_;
    int e_number_;
    double salary_;
    int dept_;
};
```

## Kommentarer till Employee

- trivial klass
  - samma överväganden som för Person
- ett Employee-objekt består av ett *subobject* av typ Person och de specifika datamedlemmarna för Employee
  - Person-subobjektet initieras före medlemmarna i Employee
  - Person-subobjektet destrueras efter att medlemmarna i Employee har destruerats
  - det enda sättet att överföra argument till en konstruktor i Person är med en *medlemsinitierare*
- båda virtuella funktionerna överskuggas
  - Employee ska ha specifika versioner av både clone() och str()
  - Employee ska vara en konkret klass – clone() *måste* överskuggas och definieras
- märk en virtuell funktion `override` (*HIC++ 10.2.1*)
  - kompilatorn kontrollerar att det verkligen finns en sådan funktion i basklassen att överskugga
  - **virtual** deklarerar *inte* (har ingen inverkan; var god kodningsstil i C++03)
- Manager är deklarerad som **friend**
  - alla medlemsfunktioner hos Manager har obegränsad åtkomst till alla medlemmar hos Employee, även **private**-medlemmar
  - *vänskap* skapar starkare koppling än härledning – härledning ger inte åtkomst till **private**-medlemmar (*HIC++ 11.2.1*)
  - anledningen till att Employee deklarerar Manager som vän sparar vi lite...

## Publik konstruktor för Employee

```
Employee::Employee(const string& name,
                   const CRN&      crn,
                   const Date&     e_date,
                   int              e_nbr,
                   double           salary,
                   int              dept)
    : Person{name, crn}, e_date_{e_date}, e_number_{e_nbr}, salary_{salary}, dept_{dept}
    {}
```

- Person-subobjektet initieras per definition först
  - Person-initieraren ska finnas först i initierarlistan
  - anrop av motsvarande konstruktor i Person
- Employees egna datamedlemmar initieras sedan i deklarationsordning
  - skriv deras initierare i samma ordningen
- en konstruktor ska uttryckligen initiera alla basklasser och icke-statiska datamedlemmar (*HIC++ 12.4.2, 12.4.4*)

## Medlemsfunktionen `str()` överskuggas

```
string Employee::str() const
{
    return Person::str() + " (Employee) " + e_date_.str() + ' ' + std::to_string(dept_);
}
```

- anropar `str()` för `Person`-subobjektet för att generera en del av strängen som ska returneras
  - kvalificerat namn – `Person::str()` – krävs för att undvika rekursion
- `std::to_string()` är överlagrad för alla grundläggande typer

## Medlemsfunktionen `clone()` överskuggas

```
virtual Employee* clone() const
{
    return new Employee{ *this };
}
```

- ska skapa ett kopia av det objekt som anropar `clone()` och returnera en pekare till kopian
- kopieringskonstruktorn är den naturliga operationen för att göra kopian
  - anropar i sin tur kopieringskonstruktorn för `Person`
- när returtypen tillhör en polymorf klasshierarki kan vi anpassa returtypen

```
Employee* p1{ new Employee{ name, crn, date, employment_nbr, salary} };
Employee* p2 = p1->clone(); // ingen typomvandling om clone() returnerar Employee*
Person* p3 = p1->clone(); // implicit typomvandling till Person* – ”upcast”
```

- typerna sägs vara *kovarianta*

## Subklassen Manager

```

class Manager : public Employee
{
public:
    Manager(const std::string& name,
            const CRN&          crn,
            const Date&         e_date,
            int                  e_number,
            double               salary,
            int                   dept);
    ~Manager() = default;

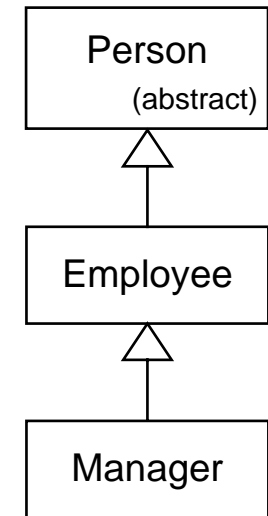
    std::string str() const override;
    Manager*    clone() const override;

    void add_department_member(Employee* ep) const;
    void remove_department_member(int e_number) const;
    void print_department_list(std::ostream&) const;
    void raise_salary(double percent) const;

protected:
    Manager(const Manager&) = default;
private:
    Manager& operator=(const Manager&) = delete;

    // Manager äger inte Employee-objekten, destruering av dessa ska inte utföras av Manager
    mutable std::map<int, Employee*> dept_members_;
};

```



## *Kommentarer till Manager*

- trivial klass
  - har en välartad datamedlem – `dept_members_` – defaultinitieras till tom map
  - för övrigt samma överväganden och åtgärder som för `Employee` och `Person`
- ett `Manager`-objekt består av ett subobjekt av typ `Employee` – som i sin tur består av ett subobjekt av typ `Person`
  - subclassobjekten – deras datamedlemmar – initieras uppifrån-och-ner och inom klasserna i deklarationsordning  

```
Person -> Employee -> Manager
```
  - destruering utförs i omvänd ordning och inom klasserna destrueras datamedlemmarna i omvänd deklarationsordning  

```
Manager -> Employee -> Person
```
- `str()` överskuggas
- `clone()` överskuggas
- `dept_members_` deklarerar **mutable**
  - `add_department_member()` och `remove_department_member()` deklarerar som **const**-funktioner av logiska skäl
  - **mutable** tillåter att `dept_members_` modifieras även av **const**-funktioner
- `Manager` är **friend** till `Employee`
  - `Manager` försöker inte komma åt privata medlemmar i `Employee` – så varför?
  - vi får strax veta...



## *Publik konstruktor för Manager*

```
Manager(const std::string& name,  
        const CRN&          crn,  
        const Date&        e_date,  
        int                 e_number,  
        double             salary,  
        int                 dept)  
    : Employee{ name, crn, e_date, e_number, salary, dept }  
    {}
```

- alla parametrar överförs som arguments till den direkta basklassen Employee
- dept\_members\_ har defaultkonstruktion – en tom lista med anställda skapas för en ny Manager

## Medlemsfunktionerna `str()` och `clone()` överskuggade

```
string Manager::str() const
{
    return Person::str() + " (Manager) " + get_employment_date().str() + ' '
        + std::to_string(get_department());
}

virtual Manager* clone() const
{
    return new Manager{ *this };
}
```

Antag att vi skulle ha glömt att överskugga `clone()` för manager:

- den *sista överskuggaren* (*last override*) vore då `Employee::clone()`
- i stället för en `Manager` skulle `clone()` returnera en `Employee`
  - kopiera av `Employee`-subobjektet i den `Manager` som skulle ha kopierats

## Anställda på avdelningen hanteras av Manager

```
void Manager::add_department_member(Employee* ep) const
{
    // Avdelningen för den anställda ska vara samma som avdelningschefens
    ep->set_department(get_department()); // vänskap behövs

    // Lägg till i avdelningens lista över anställda
    dept_members_.insert(make_pair(ep->get_employment_number(), ep));
}
```

- Manager måste vara **friend** till Employee för att få anropa **protected**-medlemmen `set_department()` i denna kontext
  - funktionsparametern `ep` är en pekare till Employee
  - endast **public**-operationer är tillåtna via `ep`, om inte Manager är vän till Employee
  - om `ep` hade varit `Manager*` hade inte **friend** behövts men det är ju inte aktuellt
- En medlemsfunktion i Manager
  - kan komma åt **private**-medlemmar i sig själv och i andra Manager-objekt
  - kan komma åt **protected**-medlemmar – egna och ärvda – i sig själv och i andra *Manager-objekt*
  - kan bara komma åt **public**-medlemmar i objekt av typ Employee och Consultant, såvida inte **friend**

## Consultant

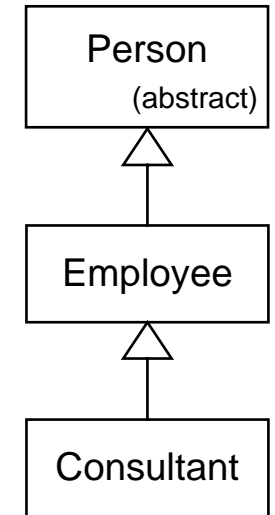
```
class Consultant final : public Employee           // ingen subclassning tillåten
{
public:
    using Employee::Employee;                       // konstruktörer ärvs

    ~Consultant() = default;

    std::string str() const override;
    Consultant* clone() const override;

protected:
    Consultant(const Consultant&) = default;

private:
    Consultant& operator=(const Consultant&) = delete;
};
```



## *Kommentarer till Consultant*

- ingen egentlig skillnad jämfört med Employee
  - samma datamedlemmar, samma operationer
  - samma publika konstruktörer ska finnas – ärvs eller genereras
- vi vill kunna särskilja konsulter från vanliga anställda
  - subtypning är ett sätt att möjliggöra det
  - görs genom dynamisk typkontroll – **dynamic\_cast** eller **typeid**-uttryck
- märka en klass `final`

```
class Consultant final : public Employee
```

- det är inte tillåtet att härleda från Consultant

- märka en virtuell funktion `final`

```
virtual void str() const override final;
```

- en sådan funktion kan inte överskuggas av subklasser
- används inte i kodexemplet

- Consultant kommer att ha två publika konstruktörer, ärvda från Employee

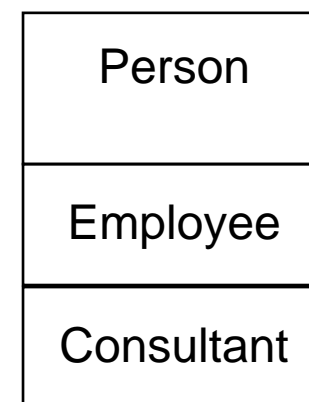
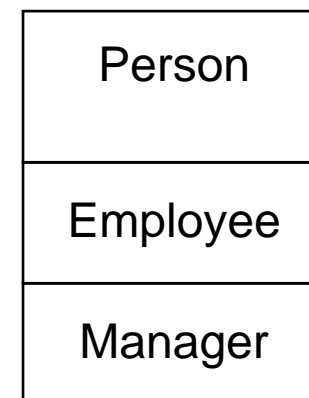
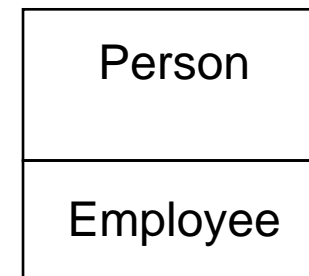
```
Consultant(const std::string&, const CRN&, const Date&, int, double);
```

```
Consultant(const std::string&, const CRN&, const Date&, int, double, int);
```

## Initiering och destruering av objekt av härledd typ

- ett objekt av härledd typ består av delar, subobjekt
  - subobjekt som motsvarar klassens basklasser
  - klassens egna datamedlemmar
- initieringsordningen är uppifrån-och ner
  - datamedlemmarna i ett basklass initieras före datamedlemmarna i en subclass
  - första konstruktorn som anropas är konstruktorn för den *mest härledda klassen*
  - den anropar sina direkta subclasskonstruktorer rekursivt
  - datamedlemmarna i en klass initieras i den ordning de deklarerats
- destrueringsordningen är omvänd mot initieringsordningen – nerifrån-och-upp
  - datamedlemmarna i en subclass destrueras före datamedlemmarna i en basklass
  - först anropas den mest härledda klassens destruktör
  - datamedlemmarna i klassen destrueras i omvänd deklaraationsordning
  - sist anropas de direkta basklassdestruktörerna, rekursivt
- viktigt att rotklassen i en polymorf klasshierarki har en virtuell destruktör
  - annars avgör pekartypen vilken destruktör som körs

```
Person* p{ new Consultant(...) };
...
delete p;           // ~Person() eller ~Consultant() ?
```



## Använda *Person*, *Employee*, *Manager*, *Consultant*

```

Person*      pp{ nullptr }; // kan peka på Employee-, Manager- eller Consultant-objekt (Person är abstrakt)
Employee*    pe{ nullptr }; // kan peka på Employee-, Manager- eller Consultant-objekt
Manager*     pm{ nullptr }; // kan endast peka på ett Manager-objekt
Consultant*  pc{ nullptr }; // kan endast peka på ett Consultant-objekt

```

```
pm = new Manager{ name, crn, date, employment_nbr, salary, 17 };
```

```
pp = pm; // "upcast" sker automatiskt – Manager* -> Person*
```

```
pm = dynamic_cast<Manager*>(pp); // "downcast" måste göras uttryckligen – Person* -> Manager*
```

```

if (pm != nullptr) // har vi en Manager?
{
    pm->print_department_list(cout);
}

```

- polymorfa pekare – kan peka på objekt som motsvarande pekarens typ och dess subtyper
- *upcast* är en automatisk och är en säker typomvandling
- *downcast* måste göras uttryckligen och kan behöva kontrolleras innan vi opererar på objektet
  - `print_department_list()` är specifik för `Manager` och kan bara anropas via pekare av typ `Manager*`
- objekten i sig förändras inte – en `Manager` är alltid en `Manager`

## Dynamisk typkontroll

Ett sätt att ta reda på ett objekts typ är med **typeid**-uttryck – inkludera `<typeinfo>`

```
if (typeid(*pp) == typeid(Manager)) ...
```

- kan användas för *typnamn*, *objekt* och alla slags *uttryck*
- ett **typeid**-uttryck returnerar ett objekt av typen `type_info`
- typkontroll kan göras genom att jämföra två `type_info`-objekt

**typeid**-uttryck:

**typeid**(\*p)     returnerar ett `type_info`-objekt för den typ av objekt som pekaren p pekar på

**typeid**(r)     returnerar ett `type_info`-objekt för den typ av objekt som referensen r refererar till

**typeid**(T)     returnerar ett `type_info`-objekt för typen T

**typeid**(p)     är vanligtvis ett misstag om p är en pekare – ger `type_info`-objekt för pekartypen

`type_info`-operationer:

`==`           testar likhet mellan två `type_info`-objekt – **typeid**(\*p) == **typeid**(T)

`!=`           testar olikhet mellan två `type_info`-objekt – **typeid**(\*p) != **typeid**(T)

`name()`       returnerar typens "namn" i form av en C-sträng – **typeid**(\*p).name()



## Dynamisk typkontroll, forts.

Typkontroll kan även göras med **dynamic\_cast**.

- användning för polymorf pekare

```
Manager* pm{ dynamic_cast<Manager*>(pp) }; // typomvandla pp

if (pm != nullptr)
{
    pm->print_department_list(cout);
}
```

– **dynamic\_cast** returnerar **nullptr** om *pp* inte pekar på ett objekt av typen `Manager` eller en subtyp till `Manager`

- användning för polymorf referens – *rp* antas ha typ `Person&`

```
dynamic_cast<Manager&>(rp).print_department_list(cout);
```

– om inte *rp* anger ett objekt av typen `Manager`, eller en subtyp till `Manager`, kastas undantaget `bad_cast`

– det finns inget ”tomma-referensen-värde” – en referens måste alltid vara bunden till ett objekt

Observera:

- med **typeid** kan man bara testa mot en *specifik* typ `T`, inte mot `T` och dess subtyper
- med **dynamic\_cast** kan man testa om ett objektet är av typ `T` eller en subtyp till `T`
- **dynamic\_cast** kräver dynamisk typinformation – endast tillåten för polymorfa klasstyper
- **typeid** kan användas för alla typer, objekt och uttryck

## *Dynamisk typomvandling*

Med operatorn **dynamic\_cast** kan man typomvandla polymorfa pekare och referenser:

**dynamic\_cast**<T\*>(p)                      *omvandlar pekaren p till "pekare till T"*

**dynamic\_cast**<T&>(r)                      *omvandlar referensen r till "referens till T"*

- "*downcast*" från *bastypspekare* till *subtypspekare* eller från *bastypsreferens* till *subtypsreferens*
- "*upcast*" är en automatisk och säker typomvandling
- vid multipelt arv förekommer även "*crosscast*"

## Sammanfattning av konstruktion av polymorfa klasshierarkier

- polymorft beteende
  - klasser måste vara polymorfa
  - objekt måste refereras av pekare eller referens
  - funktioner måste vara virtuella – anrop av en funktion som inte är virtuell binds alltid statiskt (vid kompileringen)
  - den *dynamiska typen* – typen för objektet – avgör vilken överskuggning av en virtuell funktion som anropas
- dynamisk typomvandling
  - krävs om en anropad funktion inte tillhör typen för pekaren/referensen – den *statiska typen*
  - funktionsanrop kontrolleras statiskt – inga exekveringsfel av sådan anledning
- rotklassen i en polymorf klasshierarki ska ha en *virtuell destruktör*
  - en kompilatorgenererad destruktör är vanligtvis **public** och *icke-virtuell*, men
  - om en basklass har en virtuell destruktör kommer en kompilatorgenererad subklassdestruktör också vara virtuell
- använd en virtuell medlemsfunktion som clone() för att kopiera polymorfa objekt
  - kopieringskonstruktorn bör elimineras eller deklarerar **protected** och då användas enbart internt bruk
  - kopieringstilldelningsoperatoren bör elimineras helt
- märk en klass `final` för att förbjuda subclassning
- märk en virtuell funktion `override` för att få kompilatorn att kontrollera att överskuggning görs korrekt
- märk en virtuell funktion `final` för att förhindra överskuggning i eventuella subclasser