# TDP004

## Klasser - speciella medlemsfunktioner

Eric Elfving
Institutionen för datavetenskap

LINKÖPING UNIVERSITY

- En klass ansvarar ofta för en resurs.
- Ibland är resursen så enkel som en datamedlem av inbyggd typ, men vad händer om den är svårare att hantera?
  - Fil på filsystemet
  - Nätverksanslutning
  - …

- Inom C++ brukar man använda akronymet RAII - Resource Allocation Is Initialization
- Fritt översatt betyder det att alla resurshanterande klasser ska ta resursen i sin konstruktor och sedan se till att den återlämnas när man är klar med den.

- På samma sätt som vi har konstruktorn för att initiera datamedlemmar har vi en annan speciell medlemsfunktion för att återlämna resurser - destruktorn
- Destruktorn har alltid namnet `~KLASSNAMN()` och saknar även den returntyp
- Destruktorn körs automatiskt när ett objekt av klasstypen ska försvinna (exempelvis vid slut av det block där en variabel deklarerats)

```
class My_Ptr
{
public:
    My_Ptr(int i)
      : ptr { new int{i} }
    {}
    ~My_Ptr()
    {
        delete ptr;
    }
private:
    int * ptr;
};

int main()
{
    My_Ptr p1{6}; // p1 initieras
    {
        My_Ptr p2{5}; // p2 initieras
    } // p2 destrueras
} // p1 destrueras
```

- Har vi en resurs som kräver speciell hantering för att återlämna måste vi alltid implementera destruktorn.
- I annat fall kan kompilatorn själv generera denna

# Kopiering av objekt

- Kompilatorn kan också generera stöd för kopiering av objekt utan att vi gör något speciellt

```
int main()
{
    My_Ptr t {6};
    My_Ptr t2 {t};
}
```

- Här används en kopieringskonstruktor
- Den automatgenererade varianten kommer kopiera värdet av varje datamedlem. Detta kan dock ge problem...

- Har vi behov av en destruktor har vi troligen även behov av att själva implementera kopiering.
- Vi behöver kopieringskonstruktor och kopieringstilldelningsoperator

```
class My_Ptr
{
public:
    My_Ptr(int i): ptr{ new int{i} } {}

    My_Ptr(My_Ptr const & other)  // kopieringskonstruktor
      : ptr { new int{*other.ptr} }
    {}

    My_Ptr & operator=(My_Ptr const & rhs) // kopieringstilldelning
    {
        delete ptr;
        ptr = new int { *rhs.ptr };
        return *this;  // this är en pekare till nuvarande objekt
    }
};
```

- Kopieringskonstruktorn används för att skapa nya objekt som kopior av redan existerande
- Kopieringstilldelningsoperatorn används för att ändra värde på ett existerande objekt

```
int main()
{
    My_Ptr p {5};  // vanlig konstruktor
    My_Ptr p2{p};  // kopieringskonstruktor
    p = p2;        // kopieringstilldelning
}
```

- Vi har ett problem med vår kopieringstilldelningsoperator, självtilldelning

```
    My_Ptr p {6};
    p = p;
```

- En vanlig lösning är kontroll för självtilldelning:

```
My_Ptr & My_Ptr::operator=(My_Ptr const & rhs)
{
    if ( &rhs != this )
    {
        delete ptr;
        ptr = new int { *rhs.ptr };
    }
    return *this;
}
```

- Oftast löses det smidigare med hjälp av en swap-funktion

```
void My_Ptr::swap(My_Ptr & other)
{
    std::swap(ptr, other.ptr);
}
My_Ptr & My_Ptr::operator=(My_Ptr const & rhs)
{
    My_Ptr tmp{rhs}; // skapa en kopia av rhs
    swap(tmp);       // byt innehåll med kopian
    return *this;
} // kopian (som äger mitt gamla innehåll) destrueras
```

- Detta ger idiomet kopiera och byt (copy-and-swap) eller create temporary and swap

- Jag skriver oftast enligt nedan:

```
My_Ptr & My_Ptr::operator=(My_Ptr const & rhs)
{
    My_Ptr{rhs}.swap(*this);
    return *this;
}
```

# Flyttsemantik

- Kompilatorn kan hitta många ställen där objekt behöver kopieras
- Detta kan ta mycket kraft och känns onödigt ibland
- Därför infördes flyttsemantik i C++11!
- Målet är att kompilatorn kan se när ett temporärt objekt ska kopieras och istället flytta innehållet

- Likt kopiering finns två speciella medlemsfunktioner att överlagra; flyttkonstruktor och flytttilldelningsoperator

```
class My_Ptr
{
public:
    My_Ptr(My_Ptr && other); // flyttkonstruktor, ta innehållet från other
    My_Ptr & operator=(My_Ptr && rhs); // ta innehållet från rhs
};
```

- Lägger vi till (och implementerar) dessa kan kompilatorn själv anropa dem när det passar.

# implementation

- Objektet vi tar ifrån ska lämnas i ett giltigt tillstånd

```
My_Ptr::My_Ptr(My_Ptr && other)
  : ptr { other.ptr }
{
    other.ptr = nullptr;
}
My_Ptr & operator=(My_Ptr && rhs)
{
    swap(rhs);
    return *this;
}
```

- För att testa dessa kan man använda sig av funktionen `std::move` från `<utility>`

```
My_Ptr p {5};
My_Ptr p2 { move(p) };
```

- Precis som att det är bra (och ofta nödvändigt) att lägga till `const` på funktioner som inte modifierar objektet bör man använda `noexcept` på funktioner som inte kastar undantag.

```
class My_Ptr
{
public:
    My_Ptr();
    My_Ptr(const &);
    My_Ptr(My_Ptr &&) noexcept;

    My_Ptr & operator=(My_Ptr const &);
    My_Ptr & operator=(My_Ptr &&) noexcept;

    void swap(My_Ptr &) noexcept;
};
```

- Om vi kastar ett undantag från en `noexcept`-markerad funktion dör programmet.

- Om man vill kan man även begränsa hur en viss funktion får anropas.
  - Med & i slutet av funktionsdeklarationen får funktionen endast anropas på en variabel.
  - Med && i slutet får den endast anropas på ett temporärt värde.

```
class My_Ptr
{
public:
    // ...
    My_Ptr & operator=(My_Ptr const &) &;
    // ...
};
My_Ptr fun();
int main()
{
    My_Ptr p {6};
    fun() = p; // kompileringsfel!
    p = fun(); // resultatet från fun flyttas in i p
}
```

Eric Elfving
Institutionen för datavetenskap

www.liu.se

LINKÖPING UNIVERSITY