Operatoröverlagring

En operator är i princip en funktion. Den utför en beräkning grundat på sina argument och ger ett värde som resultat. Det som skiljer en *operatorfunktion* från en vanlig funktion är dess ”namn” och hur den normalt används. För övrigt deklareras och definieras operatorfunktioner på samma sätt som vanliga funktioner, med undantag för *typomvandlingsfunktioner.* En operatorfunktion kan vara medlem i en klass, *medlemsfunktion*, eller en *vanlig funktion*, med vissa restriktioner (se avsnitt 3).

1 Definition och deklaration av operatorfunktioner

Operatorfunktioner definieras och deklareras på samma sätt som vanliga funktioner. En operatorfunktion anges med nyckelordet **operator** följt av operatorsymbolen, till exempel **operator**<<, vilket kallas för *operatorfunktionsidentifierare*. I exempel 1 visas en *definition* av **operator**<< för att skriva ut ett objekt av typen T på en utström.

```
ostream& operator<<(ostream& os, const T& t)
{
  t.print(os);
  return os;
}
```

Exempel 1 Användardefinierad överlagring av **operator**<< för en typ T.

Definitionen i exempel 1 förutsätter att T har en medlemsfunktion print, för att skriva ut ett objekt av typen T. Motsvarande *deklaration* visas i exempel 2.

```
ostream& operator<<(ostream&, const T&);
```

Exempel 2 Deklaration av operatorfunktion.

Detta sätt att deklarera **operator**<<, det vill säga valet av returtyp och parametertyper, är ett *idiom* (allmänt accepterad form) för en överlagring av **operator**<< för utskrift av en egendefinierad typ på en utström.

2 Anrop av operatorfunktioner

Operatorfunktioner anropas normalt inte direkt, utan med *operatornotation*. För en binär operator, som <<, används *infixnotation*. Om t är ett objekt av typen T, kan vi använda **operator**<< som vi har definierat i exempel 1 på följande sätt:

```
cout << t << endl;
```

Detta är helt i överensstämmelse med hur vi kan skriva ut ett värde av inbyggd typ, till exempel **int**. Det har vi sett till genom att utforma vår **operator**<< för T med eftertanke: 1) den är deklarerad som icke-medlem, 2) den har en första parameter av typ ostream& och en andra parameter av den typ vi vill skriva ut och 3) den returnerar referensen till strömmen.

För en unär prefixoperator, som negeringoperatorn -, används *prefixnotation*, -i. För en unär postfix-operator, som postfix ++, används *postfixnotation*, i++.

Det är även möjligt att anropa en operatorfunktion direkt med vanlig funktionsanropssyntax. Vi använder då operatorfunktionsidentifieraren som ”namn” och ett direkt anrop av **operator**<< skrivs alltså på följande sätt:

```
operator<<(cout, t)
```

Ett direkt anrop kan vara användbara när vi, till exempel, implementerar en operatorfunktion i termer av en annan operatorfunktion.

3 Regler för operatoröverlagring

I grunden gäller samma regler för operatoröverlagring som för överlagring av vanliga funktioner. Därutöver finns specifika regler för operatoröverlagring.

Operatorer är mest intressant att överlagra för klasstyper (**class**, **struct**, **union**) och möjligtvis uppräkningstyper (**enum**). För en klasstyp kan en operator, med några undantag, överlagras som medlemsfunktion eller som icke-medlem (vanlig funktion), eller både som medlem och icke-medlem.

När vi överlagrar en operator bör vi efterlikna motsvarande inbyggda operators egenskaper. Det innebär bland annat att välja lämplig parameteröverföring och lämplig typ för returvärdet. Det finns också en del allmänna konventioner. Om vi till exempel överlagrar + bör vi också överlagra motsvarande sammansatta tilldelningsoperator +=.

3.1 Överlagringsbara operatorer

Endast operatorsymboler som är definierade i språket kan överlagras, med undantag av `.` (punkt), `.*` , `::` och `?:`. De tre första användas för åtkomst av medlemmar i objekt och klasser och det skulle skapa problem om vi kunde definiera om dessa. De operatorer som kan överlagras visas i figur 1.

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

Figur 1 Operatorer som kan överlagras.

3.2 Olika former, prioritet och andra egenskaper

Vissa inbyggda operatorer är unära (en operand), andra är binära (två operander), några finns både i en unär och en binär form. Villkorsoperatorn ?:, som inte kan överlagras, är ternär (tre operander). Funktionsanropsoperatorn () kan ha ett godtyckligt antal parametrar. Dessa egenskaper kan inte ändras när vi överlagrar operatorner.

Både den unära och den binära formen av prefixoperatorerna +, -, * och & kan överlagras. Vilken form som avses bestäms av antalet parametrar.

Stegningsoperatorerna ++ och -- kan överlagras både i prefix- och postfixform. Postfixformerna deklareras med en extra parameter, en dummyparameter, av typ **int**.

Vi kan inte ändra på operatorernas fördefinierade prioritet, till exempel så att + skulle få högre prioritet än *. Vi kan inte heller ändra på operatorernas associativitet, till exempel så att a=b=c skulle komma att grupperas (a=b)=c, i stället för a=(b=c).

Tilldelningsoperatorn (=), unära adressoperatorn (&) och kommaoperatorn (`,`) är fördefinierade med en specifik innebörd för alla typer. De kan ändras för enskilda klasser och uppräkningstyper om vi definierar egna operatorfunktioner som implementerar dem.

Vi kan i vissa avseenden ge egna överlagringar av operatorer andra egenskaper än vad som gäller för de fördefinierade operatorerna. En egen överlagring av ++ behöver till exempel inte uppfylla att ++i är identiskt med i+=1. Inte heller behöver en egen överlagring av += förutsätta att vänsteroperanden är ett *lvalue*, vilket gäller för de inbyggda operatorerna.

I några fall kan vi inte få samma egenskaper som vissa inbyggda operatorer har. Det gäller logiskt och (&&), logiskt eller (`||`) och kommaoperatorn. För de inbyggda versionerna av dessa gäller att vänster-operanden alltid beräknas först och det är vi inte garanterade för egna överlagringar. De logiska operatorerna är kortslutande, vilket vi inte kan åstadkomma för egna överlagringar.

Då vi överlagrar en operator som ska returnera ett värde av klasstyp är det exakta valet av returtyp viktigt för att erhålla önskad semantik – ska det vara ett *lvalue* eller ett *rvalue*? Ska ett objekt eller en referens returneras? Ska returtypen vara **const** eller icke-**const**?

En operatorfunktions parametrar kan inte ha förvalda argument, med undantag av funktionsanrops-operatorn () som kan ha parametrar med förvalda argument.

3.3 Medlem eller icke-medlem?

En operatorfunktion kan antingen vara en *icke-statisk medlemsfunktion* eller en *vanlig funktion*. En egendefinierad operatorfunktion måste ha minst en parameter av klasstyp (**class**, **struct**, **union**), referens till klasstyp, uppräkningstyp (**enum**), eller referens till uppräkningtyp. För operatorerna **new**, **delete**, **new**[] och **delete**[] gäller dock andra regler, se nedan.

Operatorerna **new**, **delete**, **new**[] och **delete**[] ska vara *vanliga funktioner* eller *statiska medlems-funktioner.* Det innebär att de inte kan deklareras i annan namnrymd än global namnrymd. Om de deklareras i global namnrymd får de inte deklareras **static**. Deklarerar vi dem som medlemsfunktioner kommer de att vara statiska medlemmar, oavsett om vi deklarerar dem **static** eller ej.

Operatorerna =, (), [] och -> måste vara icke-statiska medlemmar. Det garanterar att vänsteroperanden är ett objekt av klassen i fråga. Regeln är inte nödvändig men eliminerar en del problem som annars kan uppstå på grund av att dessa operatorer vanligtvis är beroende av sin vänsteroperand och typiskt ändrar på dess tillstånd. Regeln innebär också att det inte kan finnas två olika implementeringar av operator =, dels en medlem (om inte annat genererad av kompilatorn) och dels en icke-medlem. Detta skulle, beroende på kodens struktur, kunna leda till att för identiska tilldelningsuttryck skulle i vissa fall medlemmen användas och i andra fall icke-medlemmen.

Övriga operatorer kan antingen deklareras som medlem eller icke-medlem. Vilket vi väljer kan dock vara av stor betydelse ur, till exempel, semantisk synvinkel. Det är inte heller ovanligt att man har flera optimerade överlagringar av samma operator för en viss typ.

3.4 Unära Operatorer

En unär operator ska antingen vara en icke-statisk medlemsfunktion utan parametrar eller en icke-medlem med *en* parameter, se exempel 3.

```
class X {
public:
  X operator@() const;     // Medlem
  …
};

X operator@(X);            // Icke-medlem
```

Exempel 3 Unär operator @ som medlem respektive icke-medlem.

För en prefixoperator @, kan uttrycket @x tolkas på något av följande två sätt (ej medlem till vänster, medlem till höger).

```
operator@(x)                    x.operator@()
```

Ett undantag från detta är postfix stegning (++ eller --) där ett dummyargument av typ int (0) används för att särskilja postfix och prefix stegning (ej medlem till vänster, medlem till höger).

```
operator++(x, 0)                x.operator@(0)
```

3.5 Binära operatorer

En binär operator ska antingen vara icke-statisk medlemsfunktion med *en* parameter eller en icke-medlemsfunktion med *två* parametrar, se exempel 4.


```
class X {
public:
   X operator@(X) const;   // Medlem
   …
};

X operator@(X, X);         // Icke-medlem
```

Exempel 4 Binär operator @ som medlem respektive icke-medlem.

För en binär operator @, kan uttrycket x@y tolkas på någon av följande två sätt (ej medlem till vänster, medlem till höger).

```
operator@(x, y)                x.operator@(y)
```

4 Tilldelningsoperatorer

Den vanliga tilldelningsoperatorn (=,), ska vara en icke-statisk medlemsfunktion med *en* parameter. Sammansatta tilldelningsoperatorer, som till exempel +=, behöver inte vara medlemmar. Den inbyggda tilldelningsoperatorn för **int** kan användas på följande sätt, om a, b och c är variabler av, till exempel, typen **int**:

```
a = b = c = 0
```

Förutom att den inbyggda tilldelningsoperatorn ändrar på värdet på sin vänsteroperand kan den även användas för att skriva kedjade tilldelningsuttryck, se ovan. Det senare är möjligt genom att operatorn i princip returnerar värdet på vänsteroperanden efter tilldelning. Egentligen returneras en referens till vänsteroperanden (ett *lvalue*) och den referensen kan användas antingen för att erhålla värdet på vänsterargumentet, det vanligaste, eller för att manipulera objektet som är vänsteroperand. Vi skulle kunna *tänka oss* att den inbyggda tilldelningsoperatorn för **int** definieras enligt exempel 5.

```
int& operator=(int& lhs, int rhs)
{
   // lhs tilldelas det värde som rhs har genom exempelvis medlemsvis tilldelning
   return lhs;
}
```

Exempel 5 Pseudokod för den inbyggda **operator**= för **int**.

Koden i exempel 5 anger den semantik som gäller för den inbyggda tilldelningsoperatorn för **int**. Vänsterargumentet, lhs, överförs som referens och kan därmed ändras av operatorfunktionen. Efter att värdet på lhs har ändrats returneras referensen till lhs.

Exempel på användardefinierade tilldelningsoperatorer hittar vi i std::string, se exempel 6.

```
class string {
public:
   …
   string& operator=(const string& str);
   string& operator=(string&& str);
   string& operator=(const char* s);
   string& operator=(char c);
   string& operator=(initializer_list<char> il);
   …
};
```

Exempel 6 Exempel på olika deklarationer för **operator**=.

4.1 Kopieringstilldelningsoperator

Den första tilldelningsoperatorn i exempel 6 är speciell och kallas *kopieringstilldelningsoperator.* Den motsvarar den inbyggda operator som vi skissade pseudokod för i exempel 5, men i detta fall har vi en medlemsfunktion. Vänsteroperanden är alltså underförstådd (tillgängligt via **this**) och rhs motsvaras av parametern str. Om ingen användardeklarerad tilldelningsoperator (=) finns för en klass deklareras den av kompilatorn. I exempel 7 visas deklarationen för en kopieringstilldelningsoperator.

```
T& T::operator=(const T& rhs);
```

Exempel 7 Det klassiska sättet att deklarera en kopieringstilldelningsoperator för en typ T.

Det som utmärker en kopieringstilldelningsoperator för en klass T är att den har exakt en parameter av typ T, T&, **const** T&, **volatile** T& eller **const volatile** T&. En klass kan ha flera former av kopieringstilldelningsoperatorn. Returtypen ska vara T&, ett *lvalue*. Observera att parametern inte kan ha typ T om en move-tilldelningsoperator (flytttilldelningsoperator, se nedan) deklareras.

Den kompilatorgenererade kopieringstilldelningsoperatorn gör *medlemsvis tilldelning* av klassens datamedlemmar och eventuella basklasser. Om det finns basklasser tilldelas dessa subobjekt först och sedan de icke-statiska datamedlemmarna, samtliga i den ordning de deklareras. I exempel 8 visas vad den kompilatorgenererade versionen i princip gör.

```
T& T::operator=(const T& rhs)
{
   // Tilldela (eventuella) basklassubobjekt enligt deklarationsordning
   // Tilldela klassens (direkta) icke-statiska datamedlemmar i deklarationsordning
   return *this;
}
```

Exempel 8 Principiell implementering av kompilatorgenererad kopieringstilldelningsoperator.

Varje subobjekt tilldelas på ett sätt som motsvarar dess typ. Om ett subobjekt är av klasstyp används den klassens kopieringstilldelningsoperator, om ett subobjekt är av fälttyp tilldelas varje element i fältet på ett sätt som passar elementtypen, om subobjektet är av skalär typ används en inbyggd tilldelningsoperator för den typen. Om en kopia behövs i en kopieringstilldelningsoperator och kan idiomet ”skapa en temporär och byt” med fördel kan användas för att implementera **operator**=, se exempel 9.

```
T& T::operator=(const T& rhs)
{
   T temp{rhs};                    // Skapa en kopia
   swap(temp);                     // och byt!
   return *this;
}
```

Exempel 9 Implementering av **operator**= med lokal hjälpvariabel.

Den kopia som behövs deklareras i detta fall som en lokal variabel, vilken initieras med högerargumentet. Detta kan göras med ett äkta temporärt objekt, som i exempel 10.

```
T& T::operator=(const T& rhs)
{
   T{rhs}.swap(*this);             // Skapa en temporär och byt
   return *this;
}
```

Exempel 10 Implementering av **operator**= med temporärt hjälpobjekt.

Oavsett vilken implementering vi väljer är syftet med att skriva tilldelningen på detta sätt att erhålla starkt undantagssäker kod. Om undantag kastas ska det inte uppstå minnesläckor (grundläggande undantagssäkert) och objekt ska inte hamna i ett odefinierat tillstånd (starkt undantagssäkert). En förutsättning är att medlemsfunktionen swap() kan byta värde på objekt utan att undantag kastas. Har kopian kunnat skapats utan att undantag har kastats är vi på den säkra sidan. Själva tilldelningen utförs genom byta innehåll på ***this** och kopian. Uppstädningen efter det gamla värdet för ***this** utförs då kopian destrueras. Det kan finnas effektivare sätt att skriva undantagssäker tilldelning för en viss klass och i så fall väljer man kanske hellre det, om det är motiverat.

En fråga som vi inte berört hittills är så kallad *självtilldelning*, dvs att vänster- och högerargumenten råkar vara samma objekt, dvs x = x. Om vi opererar på objekt via referenser eller pekare kan sådant inträffa utan att det framgår av koden. I bästa fall innebär felaktigt hanterad självtilldelning enbart att onödigt arbete utförs, i värsta fall att objekt hamnar i otillåtna tillstånd eller att programmet kraschar. En kontrollen av om vänster- och högerargumenen är lika kan användas i kombination med exempelvis idiomet ”skapa en temporär och byt”. Den variant av **operator**= som visades i exempel 10 kan i sådana fall modifieras enligt exempel 11.

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

Exempel 11 Implementering av kopieringstilldelningsoperator med självtilldelningskontroll.

Kostnaden för den självtest som utförs varje gång får vi väga mot att ibland i onödan skapa, initiera, byta innehåll på och destruera det temporära objektet.

4.2 Move-tilldelning

I C++11 infördes *rvalue-referenser*, T&&, och *move-semantik*. Det som tidigare betecknades enbart *referens*, T&, kallas numera *lvalue-referens*. Move-semantik (flyttsemantik) innebär att det objekt som ska ”kopieras”, källobjektet, får sitt innehåll flyttat till det objekt som ska tilldelas, destinationsobjektet, i stället för att en riktig kopia skapas. Move-semantik är i första hand tänkt för temporära objekt men kan även användas på vanliga variabler. Sådana fortlever och därför måste vara användbara även efter att deras innehåll flyttats, speciellt ska de vara *destruerbara* och *tilldelningsbara*. En lösning är att först radera destinationsobjektet, ”nollställa” det, och sedan genomföra flytten genom att byta innehåll på objekten. Källobjektet får då ett tillstånd som motsvarar defaultinitiering., vilket stämmer väl med move-semantiken.

Move-semantik kan framtvingar med hjälpfunktionen std::move(), som typomvandlar sitt argument till till rvalue-referens. Detta används numera i std::swap(), se exempel 12.

```
template<typename T>
void swap(T& a, T& b)
{
  T tmp{std::move(a)};
  a = std::move(b);
  b = std::move(tmp);
}
```

Exempel 12 Användning av hjälpfunktionen std::move().

I exempel 12 är parametrarna a och b deklarerade som lvalue-referenser. Utan move() hade kopieringskonstruktorn använts för att initiera tmp och kopieringstilldelningsoperatorn hade använts för att tilldela a och b. Så blir det också i fall T inte har någon move-konstruktor och move-tilldelningsoperator.

Exempel 13 visar ett skelett för en move-tilldelningsoperator.

```
T& T::operator=(T&& rhs)
{
   // ”Nollställ” *this
   // Flytta innehållet från rhs till *this
   return *this;
}
```

Exempel 13 Skelett för move-tilldelningsoperator.

I exempel 13 är parametern rhs deklarerad som rvalue-referens. Om vi skulle anropa en funktion i **operator**= som har en parameter som också är deklarerad som rvalue-referens måste move(rhs) användas för att överföra rhs i det anropet. rhs kommer nämligen att behandlas som en lvalue-referens, enligt *namnregeln*, ”om ett objekt har ett namn är det en lvalue-referens”. rhs kommer att kunna användas efter ett sådant funktionsanrop och då vore det inte bra om funktionen använder move-semantik utan att det uttryckligen godkänts genom att använda std::move().

En typisk implementering av move-tilldelningsoperatorn för en typ T visas i exempel 14.

```
T& T::operator=(T&& rhs)
{
   clear();
   swap(rhs);
   return *this;
}
```

Exempel 14 Implementering av move-tilldelningsoperator.

T förutsätts ha en funktion clear() som kan användas för att raderar innehållet hos vänsteroperanden och återställa till motsvarande defaultinitierat tillstånd, och en funktion swap() för att byta innehåll på två T-objekt. En enklare variant vore att enbart byta innehåll men den som visas i exempel 14 överensstämmer mer med den, antagligen, förväntade semantiken vid flytt.

5 Funktionsanropsoperatorn

Funktionsanropsoperatorn () ska vara en icke-statisk medlemsfunktion. Den kan ha ett godtyckligt antal parametrar, som kan ha förvalda argument. Båda egenskaperna är unika för funktionsanropsoperatorn. Den kan användas enligt något av följande två alternativ:

objekt(*argumentlista*) *objekt*.**operator**()(*argumentlista*)

Egentligen är *objekt* ett *postfixuttryck* som beräknas till ett klassobjekt, *argumentlista* är en lista av uttryck som motsvarar parameterlistan. Detta innebär att vi kan operera på klassobjekt på samma sätt som på funktioner. Sådana klassobjekt kallas därför *funktionsobjekt och* klassen kallas *funktionsobjektsklass*. Funktionsobjekt är lite mer generella än funktioner och de har en central roll i standardbiblioteket, tillsammans med containrar, algoritmer och iteratorer.

6 Indexeringsoperatorn

Indexeringsoperatorn [] ska vara en icke-statisk medlemsfunktion med exakt *en* parameter. Den kan användas enligt något av följande två alternativ.

objekt[*uttryck*] *objekt*.**operator**[](*uttryck*)

Egentligen är *objekt* ett *postfixuttryck* som beräknas till ett klassobjekt.

7 Klassmedlemsåtkomstoperatorn

Klassmedlemsåtkomstoperatorn -> ska vara en icke-statisk medlemsfunktion utan någon parameter. Den kan användas enligt följande:

objekt`->`*medlem*

Egentligen är *objekt* ett *postfixuttryck* som beräknas till ett klassobjekt och *medlem* anger en medlem i klassen. Tolkningen av *objekt*->*medlem* är speciell, nämligen:

(*objekt*`.`**`operator`**`->())->`*medlem*

8 Stegningsoperatorerna

Stegningoperatorerna ++ och -- kan överlagras i både prefix- och postfixform. Dessa särskiljs genom att en extra parameter, en *dummyparameter*, av typ **int** anges för postfixformerna. Dummyparametern används inte och man brukar därför inte deklarera något formellt namn för den, inte ens i funktionsdefinitionen. I ett direkt anrop anger vi ett godtyckligt värde för dummyparametern, till exempel 0. Operatornotation och motsvarande uttryckliga anrop av prefix ++ och postfix ++:

```
++i       operator++(i)
i++       operator++(i, 0)
```

Överlagrar vi prefix ++ bör vi även överlagra postfix ++, eftersom båda förväntas finnas. Det krävs eftertanke vid implementeringen, om vi ska erhålla samma semantik som för de inbyggda stegningsoperatorernas. Postfixformerna returnerar argumentets värde före stegningen i form av en kopia av detta värde (ett *rvalue*), prefixformerna returnerar det stegade värdet i form av en referens till argumentet (ett *lvalue*). Det rekommenderade sättet att implementera stegningsoperatorerna är att definiera prefixformen och sedan implementera postfixformen med hjälp av prefixformen, enligt exempel 15.

```
T& T::operator++()
{
  // Implementering …
  return *this;
}

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

```
T& T::operator--()
{
  // Implementering …
  return *this;
}

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

Exempel 15 Implementering av stegningsoperatorerna, först prefix och sedan postfix.

Av exempel 15 framgår varför prefixformen bör väljas i stället för posfixformen, då vi kan välja. Det temporära objekt som behövs i postfixformen innebär en extra kostnad jämfört med prefixformen. Detta behöver inte vara fallet för de inbyggda stegningsoperatorerna men det kan ändå vara en god vana att alltid använda prefixformen, utom i fall då postfixformen krävs.

9 Operatorerna för sammansatt tilldelning

För vissa binära operatorer, som till exempel **operator**+, finns en motsvarande sammansatt tilldelningsoperator **operator**+=. Dessa förväntas komma i par; överlagrar vi **operator**+ bör vi också överlagra **operator**+=. När vi gör det bör vi koda på ett sätt som minimerar upprepning av kod och som medför effektivtivitet. Det rekommenderade sättet att definiera **operator**@ är att implementera den i termer av **operator**@=, se exempel 16.


```
T& T::operator@=(const T&)
{
  // Implementering …
  return *this;
}

T operator@(const T& lhs, const T& rhs)
{
  T temp{lhs};
  return temp @= rhs;
}
```

Exempel 16 Överlagring av **operator**@ och motsvarande **operator**@= för en typ T.

Det vanliga är att alltså definiera **operator**@= som medlem och anledningen är att vänsterargumentet bör vara ett objekt av typen i fråga och inte något temporärt objekt som skapats genom automatisk typomvandling. Syftet med att definiera **operator**@ som icke-medlem är att möjliggöra samma typomvandlingar för både vänster- och högerargumentet. Om **operator**@ deklareras som medlemsfunktion blir automatisk typomvandling inte tillåten för vänsterargumentet. En bra lösning är att överlagra **operator**@ som icke-medlem i flera versioner. Det kan bli effektivare genom att temporära objekt då inte skapas i samband med automatisk typomvandling. Om det finns en typ U som kan typomvandlas till T, kan vi överlagra **operator**@ enligt exempel 17.

```
T operator@(const T& lhs, const T& rhs);
T operator@(const T& lhs, const U& rhs);
T operator@(const U& lhs, const T& rhs);
```

Exempel 17 Overlagring av **operator**@ för typ T när det finns en typ U som kan ”@-as” med T.

Exakt vilken parametertyp vi ska välja för U beror på vad U är för slags typ. För klasstyper väljs normalt **const**&, som i exempel 17. Flytt som alternativ till kopiering är också intressant att möjliggöra. Exempel från standardbiblioteket då det gäller överlagring av **operator**+ för std::string, se exempel 18.

```
string operator+(const string& lhs, const string& rhs);
string operator+(const string& lhs, string&& rhs);
string operator+(string&& lhs, const string& rhs);
string operator+(string&& lhs, string&& rhs);
string operator+(const string& lhs, const char* rhs);
string operator+(string&& lhs, const char* rhs);
string operator+(const char* lhs, const string& rhs);
string operator+(const char* lhs, string&& rhs);
string operator+(const string& lhs, char);
string operator+(string&& lhs, char);
string operator+(char, const string& rhs);
string operator+(char, string&& rhs);
```

Exempel 18 Överlagring av **operator**+ för std::string.

Här har vi parametrar som är av klasstyp, pekartyp och skalär typ. För std:.string gäller alltså att sammansättning av string-objekt med **operator**+ kan göras även med en C-sträng (**char***) eller med ett tecken (**char**).

9.1 Operatorerna &&, || och ,

De inbyggda operatorerna &&, || och kommaoperatorn (,) behandlas speciellt av kompilatorn:

- För samtliga gäller att vänsterargumentet alltid beräknas före högerargumentet.
- Operatorerna && och || är *kortslutande*. Högerargumentet beräknas inte om vänsterargumentets värde avgör hela uttryckets värde (**false** i fallet &&, **true** i fallet ||).
- För kommaoperatorn gäller att högerargumentet alltid beräknas efter vänsterargumentet.

Inget av detta kommer att gälla för egna överlagringar. Sådana kommer att bli vanliga funktioner i två viktiga avseenden i detta sammanhang: 1) *samtliga* argument kommer att beräknas i samband med funktionsanropen och 2) beräkningsordningen för argumenten kommer *inte* att vara bestämd.

Vi kan inte bibehålla dessa inbyggda operatorers naturliga semantik för egna överlagringar och vi kan inte skriva kod som är oberoende av beräkningsordningen för argumenten. Därmed tvingas vi bryta mot flera viktiga riktlinjer för överlagring av operatorer och dessa operatorer bör därför överlagras först efter noggrant övervägande.

10 Operatorer för hantering av dynamiskt minne

Uttryck med operatorerna **new** och **delete** är speciella, i flera avseenden. Uttrycken finns i två former, en för att hantera enstaka objekt och en för att hantera fält, se exempel 19.

```
string  ps{new string("plain new")};     // Skapa ett enstaka objekt
string* pa{new string[10]};              // Skapa ett fält med objekt

delete   ps;                             // Återlämna ett enstaka objekt
delete[] pa;                             // Återlämna ett fält med objekt
```

Exempel 19 Enkla **new**- och **delete**-uttryck.

De enkla varianterna ovan (*plain new*) kastar undantag av typen std::bad_alloc om minnestilldelningen misslyckas. Det finns en annan variant som inte kastar undantag (*nothrow new*), utan i stället returnerar 0 om minnestilldelningen misslyckas, se exempel 20.

```
string* ps{new(nothrow) string("nothrow new")};
string* pa{new(nothrow) string[10]};
```

Exempel 20 *Nothrow* **new**.

Argumentet nothrow är ett objekt av typen std::nothrow_t. En tredje variant av **new**-uttryck kan användas för att konstruerar objekt i redan befintligt minne (*placement new, in-place new*), se exempel 21.

```
char memory[sizeof(string)];                        // Tilldela minne
string* ps{new(memory) string("placement new")};    // Konstruera objekt

char memory[10*sizeof(string)];
string* pa{new(memory) string[10]};                 // Defaultkonstruktor krävs
```

Exempel 21 *Placement* **new**.

Inget nytt minne tilldelas alltså av placement-formerna, utan det man åstadkommer är i grunden att konstruera ett, eller i fältfallet, flera objekt i ett befintligt minnesutrymme.

Totalt finns det sex varianter av **new**-uttryck. Observera att sättet att deklarera memory i exempel 21 kan innebära problem till följd av krav på *alignment*. Om string-objekt endast får placeras i minnet på exempelvis startadresser som är jämnt delbara med 4, skulle annan startadress kunna medföra problem.

De **delete**-uttryck som visas i exempel 19 är de enda som man kan skriva är för att destruera objekt och återlämna minne; det finns alltså inga **delete**-uttryck som motsvarar *nothrow*- eller *placement*-formerna av **new**-uttryck. De fungerar för objekt som skapats med både *plain new* och *nothrow new*. Om man har använt **new** för att skapa ett objekt ska man använda **delete** för att återlämna det; har man använt **new**[] ska man använda **delete**[]. Objekt som skapas med *placement new* destruerar man genom att uttryckligen anropa destruktorn (kanske det enda tillfället som man behöver göra det), se exempel 22.

```
ps->~string();                                  // Uttryckligt destruktoranrop

for (string* ps = pa; ps != pa + 10; ++p)       // För objekt i ett fält
  p->~string();
```

Exempel 22 Uttryckligt anrop av destruktor.

Om och hur själva minnet ska hanteras i *placement*-fallet beror på hur det erhållits från början.

10.1 Operatorfunktionerna new och delete

Lite förvillande kan vara att det finns en uppsättning *operatorfunktioner* i standardbiblioteket som också heter `new` och `delete`. De hänger ihop med **new**- och **delete**-uttryck men till skillnad från andra operatorfunktioner överlagrar *inte* dessa biblioteksfunktioner **new** och **delete** i uttrycken. De anropas i stället som ett led då uttrycken utförs och ansvarar endast för minneshanteringen. Följande **new**-uttryck skapar och initierar ett string-objekt.

```
string* ps{new string("C++")};
```

Uttrycket utförs i tre steg. Först anropas biblioteksfunktionen **operator** `new` för att tilldela nytt minne som är stort nog att rymma objektet. Därefter körs konstruktorn för string för att initiera objektet med det initialvärde som givits, ”C++”. Om inget initialvärde ges körs defaultkonstruktorn. Slutligen returneras en pekare till objektet.

För att destruera och återlämna minnet för ett dynamiskt objekt används ett **delete**-uttryck, se nedan.

```
delete ps;
```

Detta utförs i två steg. Först körs destruktorn på objektet och sedan anropas biblioteksfunktionen **operator** `delete` för att återlämna minnet.

Egenskaperna hos **new**- och **delete**-uttrycken kan inte ändras. Däremot kan standardbibliotekets operatorfunktioner överlagras eller deklareras som medlemsfunktioner. I det senare fallet döljer de standardbibliotekets funktioner i **new**- och **delete**-uttryck som opererar på klassen i fråga.

```
void* operator new(size_t size);
void  operator delete(void* ptr) noexcept;
void* operator new[](size_t size);
void  operator delete[](void* ptr) noexcept;

void* operator new(size_t size, const nothrow_t&) noexcept;
void  operator delete(void* ptr, const nothrow_t&) noexcept;
void* operator new[](size_t size, const nothrow_t&) noexcept;
void  operator delete[](void* ptr, const nothrow_t&) noexcept;

void* operator new(size_t size, void* ptr) noexcept;
void  operator delete(void* ptr, void*) noexcept;
void* operator new[](size_t size, void* ptr) noexcept;
void  operator delete[](void* ptr, void*) noexcept;
```

Exempel 23 Standardbibliotekets operatorfunktioner new och delete.

I exempel 23 ovan visas deklarationerna för standardbibliotekets operatorfunktioner new och delete. Namnen skrivs inte med fet stil för att markera att funktionerna inte överlagrar **new** och **delete** i uttryck. De fyra första är de vanliga versionerna, de fyra nästa är nothrow-versionerna och de fyra sista placement-versionerna.

Operatorfunktionerna, förutom *placement*-varianterna, används för att tilldela och återlämna dynamiskt minne. När *placement*-formerna används erhålls minnet på annat sätt och det är de andra stegen i beräkningen av **new**-uttrycken man är ute efter. Observera att då det gäller delete-funktionerna är det bara två som motsvarande **delete**-uttryck som kan skrivas i program. De övriga finns främst för att kompilatorn kan komma att anropa dem. Det som sägs för new och delete nedan gäller även för fältformerna new[] och delete[].

Standardbibliotekets operatorfunktioner är globala, de tillhör *inte* standardnamnrymden std. Vill man använda några andra än de enkla formerna ska man inkludera <new>. *Placement*-formerna går inte att ersätta, däremot kan de överlagras som medlemmar av klasser.

För att få lite mer förståelse för hur bibliotekets operatorfunktioner och **new**-uttryck hänger ihop kan följande enkla **new**-uttryck hjälpa till.

```
string* ps{new string("plain new")};
```

Detta ersätts i princip med följande kod av kompilatorn:

```
void* tmp{::operator new(sizeof(string))};      // Tilldela minne
string* ps;
try
{
  ps = new(tmp) string("plain new");            // Konstruera med placement new
}
catch (...)
{                                               // Om konstruktionen misslyckas,
  ::operator delete(tmp, sizeof(string));       // återlämna minnet
}
```

Först tilldelas minne med biblioteksfunktionen **operator** new och refereras tills vidare med en hjälppekare tmp. Sedan deklareras användarens pekare ps, utan att initieras. Därefter används *placement* **new** för att konstruera string-objektet i minnet som hjälppekaren pekar på. Skulle det kastas ett undantag under konstruktionen av string-objektet, fångas det in och minnet återlämnas av bibliotekfunktionen **operator** delete. Notera att kompilatorn väljer en delete-funktion som motsvarar den new-funktion som används för att tilldela minne, dvs bortsett från den första parametern till delete kommer övriga parametrar att överensstämma. När man överlagrar en viss form av new ska man därför även överlagra motsvarande form av delete, även om den inte anropas av användare. Kompilatorn kan alltså efterfråga en sådan överlagring och om den inte finns kan det leda till en minnesläcka eller andra problem. Varje klasspecifik överlagring av new, som t.ex.

```
static void* operator new(parametrar);
```

ska alltså åtföljas av en motsvarande överlagring av delete, d.v.s.

```
static void operator delete(void*, parametrar);
```

Parametrar är en lista av extra parametertyper, där den första alltid ska vara std::size_t.

Minneshanteringsfunktioner kommer alltså i par, se exempel 24. Om man för en klass tänker definiera *någon* variant av new bör samtliga tre standardformer överlagras, annars förloras de inbyggda former som inte överlagras. Det beror på att ett namn som deklareras i ett visst deklarationsområde, i detta fall i en klass, alltid döljer samma namn i omgivande deklarationsområden och att överlagring aldrig sker mellan deklarationsområden.


```
void* operator new(size_t size);                                  // vanliga
void  operator delete(void* ptr) noexcept;

void* operator new(size_t size, const nothrow_t&) noexcept;   // nothrow
void  operator delete(void* ptr, const nothrow_t&) noexcept;

void* operator new(size_t size, void* ptr) noexcept;          // placement
void  operator delete(void* ptr, void*) noexcept;
```

Exempel 24 De tre standardformerna av new och deras tillhörande delete-former.

Minneshanteringsfunktionerna är alltid *statiska* medlemsfunktioner, även om de inte uttryckligen deklareras med **static**, se exempel 25.

```
class C {
public:
  …
  static void* operator new(std::size_t);
  static void  operator delete(void* ptr) noexcept;
  static void* operator new(std::size_t, const std::nothrow_t&) noexcept;
  static void  operator delete(void* ptr, const nothrow_t&) noexcept;
  static void* operator new(std::size_t, void*) noexcept;
  static void  operator delete(void* ptr, void*) noexcept;
  …
};
```

Exempel 25 Operatorfunktionerna new och delete som klassmedlemmar.

Vill man förhindra att objekt för en klass tilldelas minne med vanliga new eller *nothrow* new**,** kan man deklarera dem i klassens **private**-del. *Placement* new bör normalt inte döljas, eftersom den används flitigt av standardbibliotekets containerklasser.

10.2 Typomvandlingsfunktioner

En medlemsfunktion som har en deklaration på formen ’**operator** *conversion-type-id*()’ specificerar en typomvandling från klassen i fråga till typen *conversion-type-id*, se exempel 26.

```
class Wrapper {
public:
  Wrapper(const int i = 0) : value_(i) {}
  operator int() const { return value_; }
private:
  int value_;
};

Wrapper w{7};

int i{w};                       // direktinitiering
int i(w);                       // direktinitiering, alternativ syntax
int j = w;                      // kopieringsinitiering, implicit typomvandling
int k = int(w);                 // explicit typomvandling, funktionsform
int m = (int)w;                 // explicit typomvandling, ”cast notation”
int n = static_cast<int>(w);    // explicit typomvandling, static_cast
```

Exempel 26 Typomvandlingsfunktion för omvandling från C till **int**.

En typomvandlingsfunktion kan inte ha några parametrar och inget returvärde och ska alltså vara en medlemsfunktion. Typomvandlingsfunktioner ärvs, kan vara **virtual** men inte **static**. I C++11 kan typomvandlingsfunktioner deklareras **explicit** och kan då endast användas explicit i samband med direktinitiering (initieringen av j i exempel 26 skulle alltså *inte* vara tillåten).

11 Riktlinjer för operatoröverlagring

Vi bör ha goda skäl för att överlagra en operator. Användningen ska ha en klar intuitiv tolkning, annars kan det vara tecken på missbruk av möjligheten att överlagra. Till exempel bör en överlagring av + innebära att, i någon naturlig mening, ”addera” eller ”sätta samman” två värden av typen i fråga.

Om det uppstår oklarheter om semantiken för en överlagrad operator är det vanligtvis bättre att välja en vanlig funktion, eftersom det inte finns motsvarande förväntningar på funktioner. Det finns alltid undantag från allmänna regler och det kan i speciella fall finnas motiv för att överlagra operatorer, trots att semantiken avviker påtagligt från den normalt förväntade.

Vissa operatorer förväntas förekomma i grupp. Kan vi skriva a+b ska vi också kunna skriva a+=b. I vissa fall är det naturligt att förvänta sig att även en ”invers” finns. Exempel på sådana ”inverser” är + och - eller >> och <<. Symmetri bör eftersträvas för vissa operatorer. Kan vi skriva a+b, ska vi också kunna skriva b+a.

Då vi bestämt oss för att överlagra en operator, bör vi ta reda på vad som gäller för motsvarande inbyggda operator och efterlikna det så långt möjligt. En tumregeln är att göra som det fungerar för **int**. Några exempel följer nedan.

Den inbyggda binära operatorn + tar två värden (*rvalue*) som argument och ger ett värde (*rvalue*) som resultat. *I princip* bör en egen överlagring deklareras enligt följande:

```
T operator+(const T, const T);
```

För en egendefinierad typ T är dock parametertypen *referens till konstant*, **const** T&, vanligtvis att föredra. En egendefinierad överlagring av + bör därför normalt deklareras som:

```
T operator+(const T&, const T&);
```

Den inbyggda **operator**= tar adressen (*lvalue*) till vänsteroperanden och värdet (*rvalue*) av högeroperanden som argument och ger adressen till vänsteroperanden som resultat (*lvalue*). En egendefinierad överlagring bör därför normalt väljas som:

```
T& operator=(const T&);
```

I vissa situationer kan argument typomvandlas automatiskt, till exempel om vi adderar ett **char**-värde och ett **int**-värde. Det kan innebära att vi överlagrar enligt följande om vi har två typer T och U som det är naturligt att addera i någon mening och få ett resultat av typ T:

```
T operator+(const T&, const T&);
T operator+(const T&, const U&);
T operator+(const U&, const T&);
```

Se avsnittet 12 för en översikt av de inbyggda operatorerna.

11.1 Medlem eller icke-medlem

I exempel 27 nedan anges riktlinjer för att bestämma om en operatorfunktion måste eller bör vara medlem eller ej samt om den ska vara **friend** eller **virtual**.

Om operatorn är en av `.`, `.*`, `::` eller `?:` kan den *inte* överlagras.

Om operatorn är en av `=`, `->`, `[]` eller `()` *måste* den vara **medlem**.

Om operatorn

a) kan ha en annan typ som vänsterargument, eller

b) kan ha typomvandling för sitt vänsterargument, eller

c) kan implementeras med enbart hjälp av klassens publika gränssnitt,

gör den till **icke-medlem** och, om så behövs i fall **a** och **b**, till **vän** (**friend**).

Om den behöver bete sig **virtuellt**,

lägg till en virtuell medlemsfunktion och implementera funktionen i termer av den.

Annars, låt operatorn vara **medlem**.

Exempel 27 Allmänna riktlinjer för överlagring av operatorfunktioner.

Kommentarer till riktlinjerna i exempel 27:

- Fall **a)** gäller till exempel för operatorerna >> och << och strömmar. Om vi vill efterlikna vad som gäller för inbyggda typer, inklusive möjligheten att skriva kedjade uttryck, kan inte dessa överlagras som medlemsfunktioner.
- Fall **b)** kan till exempel gälla operatorn +. Om vi överlagrar + för en typ T och en annan typ U har en användardefinierad typomvandling till T, kan typomvandling kombineras med + i uttryck som t+u och u+t, där t är av typ T och u är av typ U. Fallet u+t vore ej tillåtet om + överlagras som en medlem.
- Ett exempel på virtuellt beteende är om vi vill överlagra operator << för utskrift av objekt tillhörande en klasshierarki. Om basklassen i hierarkin är Base, överlagrar vi << för **const** Base&, som kan bindas till objekt av typ Base samt dess subklasser.
- Ett bra skäl för att överlagra en operator som medlem är till exempel om vi endast vill tillåta att den används om vänsterargumentet är av typen i fråga. Det skulle kunna gälla en sammansatt tilldelningsoperator som +=, där vi inte vill tillåta möjligheten att automatiskt typomvandla vänsterargumentet (och erhålla ett temporärt objekt) innan operatorn appliceras. Stegningsoperatorerna ++ och -- är också starka kandidater till att vara medlemmar.

Det är inte ovanligt att överlagra en operator både som medlem och som icke-medlem och även i flera versioner för olika parametertyper. Optimerade versioner för olika typkombinationer kan ofta vara ett bättre alternativ ur effektivitetssynvinkel, än att ha en version av operatorn som kan kombineras med automatisk typomvandling.

Fler riktlinjer för operatoröverlagring ges nedan. En del av dessa gäller även för funktioner i allmänhet.

- Bevara operatorers naturliga semantik. Om det inte är möjligt bör vanliga funktioner övervägas som alternativ.
- Parametrisera väl. Skilj på in-, ut- och in/ut-parametrar och mellan värde- och referensparametrar. Detta leder både till säkerhet och effektivitet.

  För inparametrar:

  – **const**-deklarera alltid pekare och referenser, såvida inte speciell semantik föreligger som kräver annat, t.ex. destruktiv kopiering.

  – överför objekt av grundläggande typ och objekt som är billiga att kopiera som värde.

  – överför övriga typer av objekt som **const**&.

  – överväg värdeöverföring i stället för referensöverföring om funktionen behöver en kopia av ett argument.

För ut- och in/ut-parametrar:

– föredra att överföra argument via pekare om argumentet är valfritt. Anrop kan då göras med 0 för att representera fall som innebär ”ej tillgänglig” eller ”ej relevant”.

– föredra att överföra via referens om argumentet krävs och funktionen inte lagrar en pekare till argumentet eller på annat sätt kan påverka ägarskapet. Detta visar att argumentet krävs och att anroparen måste leverera ett användbart objekt.

- Om vi överlagrar en operator @ som har en motsvarande tilldelningsoperator @=, ska vi överlagra även den sammansatta operatorn och helst implementera @ i termer av @=, se avsnitt 9.
- Returnera alltid strömreferenser från operator << och >>, då de överlagras för strömmar.
- Implementera alltid prefix och postfix stegning (++ eller --) i par och implementera postfix stegning i termer av prefix stegning, enligt riktlinjerna som ges i avsnitt 8.
- Undvik att överlagra operatorerna &&, || och , (komma), se avsnitt 9.1.
- Se upp med dolda temporära objekt. Överväg till exempel att överlagra en operator i flera versioner för att undvika temporära objekt som kan uppstå vid automatisk typomvandling av operander.

12 Inbyggda operatorer och deras semantik

När man överlagrar en operator bör man normalt efterlikna den inbyggda operatorn, då det gäller parametertyp(er), returtyp och övriga egenskaper. För några av de inbyggda operatorerna (&&, || och ,) kan man inte åstadkomma detta fullt ut (se 9.1 Operatorerna &&, || och ,).

Här ges en översikt över vad som normalt bör gälla vid överlagring av operatorer då det gäller parametertyper och returtyper. Detta grundas på vad som gäller för de inbyggda operatorerna. En del saker, t.ex. variationer med avseende på **const**- och **volatile**-kvalificering av parameter- och returtyper har utelämnats och genomgången är inte helt uttömmande (pekare-till-medlem-operatorn ->* tas inte upp).

Stegningsoperatorerna ska ha en operand som är ett ändringsbart *lvalue*. *Prefixvarianterna* ska returnera ett ändringsbart *lvalue*. Egna överlagringar kan deklareras enligt nedan för en typ T (ej pekartyp):

```
T& operator++(T&);          T& operator--(T&);
```

Postfixvarianterna ska returnera ett *rvalue*. Egna överlagringar kan deklareras enligt följande med det dummy-argument som används för att ange postfixvarianterna:

```
T operator++(T&, int);      T  operator--(T&, int);
```

För pekarttyper kan egna överlagringar av stegningsoperatorerna deklareras enligt följande för en typ T:

```
T& operator++(T*&);         T& operator--(T*&);
T  operator++(T*&, int);    T  operator--(T*&, int);
```

Avrefereringsoperatorn * (*indirection*) tar ett argument av pekartyp och returnerar ett *lvalue*, som refererar till det objekt som pekas på. En egen överlagring för en typ T kan deklareras enligt följande:

```
T& operator*(T*);
```

Unära plusoperatorn + tar ett värde och returnerar ett värde (för de inbyggda operatorerna samma värde som argumentet). En egen överlagring för en typ T kan deklareras enligt följande:

```
T operator+(T);             T* operator+(T*);
```

Negeringsoperatorn - tar ett värde och resultatet är ett värdet (för de inbyggda operatorerna det negerade värdet av argumentet). En egen överlagring för en typ T kan deklareras enligt följande:

```
T operator-(T);
```


De **multiplikativa operatorerna** *, / och %, och de **additiva operatorerna** + och - tar två värden som argument och returnerar ett värde. För de inbyggda operatorerna kan operanderna vara av aritmetisk typ, utom för operatorn % som ska ha argument av heltalstyp. Operanderna kan vara av olika typ och typomvandling kan ingås. Egna överlagringar för en typ T kan deklareras enligt följande:

```
T operator*(T, T);          T operator/(T, T);
T operator%(T, T);

T operator+(T, T);          T operator-(T, T);
```

Bitoperatorena & (OCH), | (ELLER) och ^ (exklusivt ELLER, XOR) tar två värden och returnerar ett värde. Argumenten kan vara av heltalstyp och typomvandling kan ingå. Egna överlagringar för en typ T kan deklareras enligt följande:

```
T operator&(T, T);
T operator|(T, T);
T operator^(T, T);
```

Skiftoperatorerna << och >> tar två värden och returnerar ett värde som är vänsterargumentets värde skiftat det antal steg som högerargumentet anger. Argumenten kan vara av olika heltalstyper och resultatet är av samma typ som vänsterargumentet. Egna överlagringar för en typ T kan deklareras enligt följande:

```
T operator<<(T, R);         T operator>>(T, R);
```

Ett-komplementsoperatorn ~ tar ett värde och returnerar värde. För de inbyggda operatorerna kan argumentet vara av heltalstyp (*integral type*) och resultatet är ett-komplementet av argumentet. Typoomvandling av argumentet kan ingå. En egen överlagring för en typ T kan deklareras enligt följande:

```
T operator~(T);
```

Relationsoperatorerna (<, <=, >, >=) och **likhetsoperatorerna** (== och !=) tar två värden och returnerar **bool**. För de inbyggda operatorerna kan argumenten vara av heltalstyp, pekartyp eller uppräkningstyp. De två argumenten behöver inte vara av samma typ då det gäller heltalstyper och typomvandling kan då ingå. Egna överlagringar för en typ T kan deklareras enligt följande:

```
bool operator<(T, T);       bool operator>(T, T);
bool operator<=(T, T);      bool operator>=(T, T);

bool operator==(T, T);      bool operator!=(T, T);
```

De logiska operatorerna ! (logisk negation), && (logiskt OCH) och || (logiskt ELLER) tar två **bool**-värden och returnerar ett **bool**-värde. De inbyggda operatorernas deklarationer är alltså följande:

```
bool operator!(bool);
bool operator&&(bool, bool);
bool operator||(bool, bool);
```

Skulle man vilja överlagra dem för någon typ T skulle det innebära att byta **bool** mot T. Den bestämda beräkningsordningen för argumenten och den kortslutande effekt som de inbyggda operatorerna && och || har kan inte erhållas för egna överlagringar.

- För varje objekttyp T:

```
T* operator+(T*, std::ptrdiff_t);
T* operator+(std::ptrdiff_t, T*);

T& operator[](T*, std::ptrdiff_t);
T& operator[](std::ptrdiff_t, T*);

T* operator-(T*, std::ptrdiff_t);
```


Tilldelningsoperatorn = och de **sammansatta tilldelningsoperatorerna** @=, där @ är någon av de aritmetiska operatorerna *, /, +, -, %, <<, >>, &, | eller ^ , tar ett ändringsbart *lvalue* som vänsterargument, ett värde (*rvalue*) som högerargument och returnerar en referens till vänsterargumentet (ett ändringsbart *lvalue*).

Den inbyggda operatorn = kan ta ett vänster- och högerargument där båda är av aritmetisk typ (kan vara olika aritmetiska typer och högerargumentet kan typomvandlas vid behov), uppräkningstyp, pekare till godtycklig typ eller pekare-till-medlem-typ.

De inbyggda operatorerna =, +=, -=, *= och /= kan ta argument av aritmetisk typ, och %=, <<=, >>=, &=, |= och ^= kan ta argument av heltalstyp. För högerargumentet kan typomvandling ingå. Operatorerna += och -= kan dessutom ta ett vänsterargument av pekartyp och ett högerargument av heltalstyp (samma som standardtypen `ptrdiff_t`).

Egna överlagringar av tilldelningsoperatorerna för en typ T kan deklareras enligt följande:

```
T& operator=(T&, T);
T& operator*=(T&, T);        T& operator/=(T&, T);
T& operator%=(T&, T);
T& operator+=(T&, T);        T& operator-=(T&, T);

T& operator<<=(T&, T);       T& operator>>=(T&, T);
T& operator&=(T&, T);        T& operator|=(T&, T);
T& operator^=(T&, T);
```

