

## Operatoröverlagring

- endast operatorsymboler definierade i C++ kan överlagras

```

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

```

- fyra operatörer är *inte* tillåtna att överlagras

```

.   .*   ::   ?:

```

- de tre första är fundamentala för åtkomst och deklaration av klassmedlemmar
- ?: har en strikt ordning för beräkning av argumenten som inte kan specificeras för en egen överlagring

- fyra operatörer måste vara en *icke-statisk medlemsfunktion*

```

=   ( )   [ ]   ->

```

- säkerställer att vänsteroperanden är ett objekt av typen ifråga

- övriga kan antingen vara en

- icke-statisk medlemsfunktion med *ingen* parameter (unär operator) eller *en* parameter (binär operator) – **this** finns implicit
- icke-medlemsfunktion med *en* parameter (unär operator) eller *två* parametrar (binär operator)

## Operatoröverlagring, forts.

- anropssyntaxen kan väljas

- som vanligt för operatörer (infix-, prefix- eller postfix-notation)

```
a + b
```

- om medlem – vanligt medlemsfunktionsanrop

```
a.operator+(b)
```

- om icke-medlem – vanligt funktionsanrop

```
operator+(a, b)
```

- prioritet och associativitet gäller enligt de inbyggda operatörerna

- beräkningsordningen för argument kan *inte* styras

- medför problem vid överlagring av exempelvis && och || – överlagras inte sådana operatörer (*HIC++ 13.2.1*)

- välj parametertyper och resultattyp med omsorg

- se till att returtypen för en binär operator matchar dess inbyggda motsvarighet (*HIC++ 13.2.2*)

- deklarerar binära aritmetiska operatörer och bitvisa operatörer som icke-medlemmar (*HIC++ 13.2.3*)

- tänk på att icke-medlemsfunktioner ska deklaras i samma namnrymd som typen de tillhör (String i detta fall)

- ADL (Argument Dependent Lookup) kan annars ställa till det – funktionen hittas inte eller, i värsta fall, väljs fel funktion

## Indexeringsoperator

- `operator[]` gör ingen kontroll av indexvärdet – ska alltid implementeras med en icke-`const`- och en `const`-version (*HIC++ 13.2.4*)

```

char& String::operator[](int pos)           // för icke-const String
{
    return p_[pos];
}

char String::operator[](int pos) const     // för const String
{
    return p_[pos];
}

```

- motsvarande medlemsfunktion `at()` kontrollerar om given position (index) är tillåten – om inte kastas undantaget `out_of_range`

```

char& String::at(int pos)                  // för icke-const String
{
    if (pos < 0 || pos >= size_)
        throw out_of_range{"String::at()"};
    return p_[pos];
}

char String::at(int pos) const            // för const String
{
    if (pos < 0 || pos >= size_)
        throw out_of_range{"String::at()"};
    return p_[pos];
}

```

## Strängsammansättning

```
s1 += s2;
```

```
s3 = s1 + s2;
```

- sammansättning av två String-objekt med +=

```

String& String::operator+=(const String& rhs)
{
    if (!rhs.empty()) append_(rhs.p_);
    return *this;
}

```

- sammansättning av två String-objekt med +

```

String operator+(const String& lhs, const String& rhs)
{
    return String{ lhs }.operator+=(rhs);
}

```

- operatörer som + och += hör ihop – finns den ena förväntas den andra också finnas

- `operator+` implementeras med hjälp av `operator+=` (*HIC++ 13.2.5*)
- säkerställer konsekvent semantik för += och +
- exempel på när man använder vanlig medlemsfunktionsanropssyntax för operatorfunktion

## Utskrift med operator<<

```
String s{"foobar"};

cout << s << endl;
```

- `operator<<` kan inte vara medlem – vänster operand är `ostream`, inte `String`
- standardbibliotekets `operator<<` för `const char*` kan användas för att implementera
- i och med att den publika medlemsfunktionen `c_str()` finns behöver `operator<<` inte vara **friend**

```
ostream& operator<<(ostream& os, const String& str)
{
    return os << str.c_str();
}
```

## Överlagrade varianter

Varianter för jämförelse med `char*`:

```
bool operator==(const String& lhs, const String& rhs);

bool operator==(const String& lhs, const char* rhs);

bool operator==(const char* lhs, const String& rhs);
```

Detta tillåter följande jämförelser utan typomvandling (och därmed att temporära objekt skapas):

```
String s1{"foo"};
String s2{"fie"};

char s3[] = "fum";

s1 == s2

s2 == s3

s3 == s1
```

## Friend eller inte?

En vanlig operatorfunktion kan vara vän, **friend**.

```
class String
{
public:
    ...
    friend bool operator==(const String& lhs, const String& rhs);
    ...
};

bool operator==(const String& lhs, const String& rhs);
```

Undvik om möjligt.

- vänskap är den starkaste formen av koppling man kan åstadkomma till en klass.
  - ger tillgång till alla delar av klassen, även privata
  - arv ger inte tillgång till klassens privata delar
- vänskap går att undvika om det finns publika medlemsfunktioner som kan användas:

```
bool operator==(const String& lhs, const String& rhs)
{
    return strcmp(lhs.c_str(), rhs.c_str()) == 0;
}
```

## Operatoröverlagring – riktlinjer (se även HIC++ 13.2) †

- **om** operatören är en av följande kan den inte överlagras

```
.      .*      ::      :?
```

- **om** operatören är en av följande måste den vara *medlem*

```
=      ->      [ ]      ( )
```

- **om** operatören

1. kan ha en annan typ som vänsterargument, eller den
2. kan ha typomvandling för sitt vänsterargument, eller den
3. kan implementeras enbart med publika medlemsfunktioner,

gör den till *icke-medlem* och, om nödvändigt i fall 1 och 2, även **friend** om implementeringen kräver åtkomst till icke-publika medlemmar

- **om** operatören behöver bete sig virtuellt, lägg till en virtuell medlemsfunktion och låt operatören anropa den medlemsfunktionen (av intresse i samband med polymorfa klasser)

- **i annat fall**, låt operatören vara *medlem*

– det finns dock fler operatörer som det vanligtvis är naturligt att (vänster)operanden ska vara ett objekt av typen ifråga:

```
*=      /=      %=      +=      -=      &=      |=      ^=      <<=      >>=      ++      --
```

- överlagra **inte** operatörer med speciell semantik, dvs `&&`, `|` och `,` (kommaoperatören) (HIC++ 13.2.1)

– samtliga vänsterassociativa, högeroperanden till `&&` och `|` ska enbart beräknas om vänsteroperanden beräknats **true/false**

- överlagring av adressoperatören `&` kan leda till odefinierat beteende om den används på ställe där den egna inte är synlig

## Sammanfattning

Vi har studerat en icke-trivial klass och i samband med den tagit upp ett antal viktiga saker.

- initiering – konstruktörer av olika slag
  - defaultkonstruktor
  - kopieringskonstruktor och flyttkonstruktor
  - andra konstruktörer, bland annat typomvandlande
- destruering
  - String-objekt i sig återtas automatiskt då deklaraionsblocket lämnas eller **delete** utförs på ett dynamiskt String-objekt
  - det tillhörande dynamiska minnet måste uttryckligen återlämnas – **delete** [ ] p\_
- tilldelning
  - kopieringstilldelning och flyttilldelning
  - andra tilldelningar, bland annat typomvandlande
- olika operationer
  - överlagrade operatörer och vanliga funktioner
  - medlem eller icke-medlem?
  - swap-funktioner är mycket användbara för containerliknande klasser
  - mycket viktigt att deklarerar medlemsfunktioner som inte ändrar på datamedlemmar **const** (HIC++ 9.1.1)
  - alltid viktigt att använda **const** för saker som inte ska ändras – objekt, medlemsfunktioner, parametrar,...
- iteratorer
  - visade sig enkelt i detta fall (**char\***)

## Sammanfattning, forts.

- vi har lyckats undvika att vän-deklarera (**friend**) icke-medlemmar (HIC++ 11.2)
  - publika medlemsfunktioner som ändå ska finnas används
- typomvandling tillåts under kontrollerade former
  - behovet av implicit typomvandling har minimerats – tumregeln säger annars att det bör elimineras
  - vissa binära operatörer har överlagrats i versioner som kan ta String och **char\*** blandat
  - förekomsten av temporära objekt minimeras
  - implicit typomvandling från **char\*** till String tillåts – får anses problemfri och användbart
  - endast explicit typomvandling till **char\*** tillåts (HIC++ 12.1.1) – eftersom det är en pekartyp skulle annars mycket kunna hända...
- in- och utmatning
  - funktionalitet och implementation i analogi med inbyggda datatyper
- undantagssäker programmering
  - genomtänkt kodning i situationer då undantag kan kastas
  - inga objekt blir defekta
  - inget minne läcker
- användbart idiom – *skapa en temporär och byt* (HIC++ 12.5.6)
- uppsättningen operationer behöver utvidgas för att få en användbar strängtyp