Klasser

Ett posttyp – **class** eller **struct**:

- komponenterna kallas **medlemmar** – grundläggande är *datamedlemmar* och *medlemsfunktioner*
- **struct** – används lämpligtvis då alla medlemmar ska vara allmänt åtkomliga
- **class** – används då en del medlemmar inte ska vara direkt åtkomliga
- *klasstyp* är en gemensam beteckning på **struct**, **class** och **union** (den senare tas inte upp i kursen)

Har modulegenskaper:

- kapslar in sina medlemmar
- medlemmarna tillgänglighet styrs med *åtkomstspecificerare* och **friend**-deklaration
  - **public** – allmänt tillgängliga – default för **struct**
  - **private** – endast tillgängliga för klassens egna medlemsfunktioner – default för **class**
  - **protected** – tillgängliga endast för subklassers medlemsfunktioner – används vid härledning/arv
  - **friend** – alla medlemmar i klassen blir fritt tillgängliga för en funktion, en medlemsfunktion i annan klass eller en klass (alla medlemsfunktionerna i den klassen) – undvik!

Operationerna på en klasstyp utgörs av:

- klassens medlemsfunktioner
- vanliga funktioner som har parametrar av klasstypen i fråga och ingår i klassens naturliga operationer
- fördefinierade operationer, speciellt *punktoperatorn* och *piloperatorn* för åtkomst av medlemmar

Klassmedlemmar

Klassmedlemmar kan vara av fyra slag:

- *datamedlemmar* – variabler och konstanter som deklareras inuti klassen
- *medlemsfunktioner* – funktioner som deklareras inuti klassen
- *nästlade typer* – typdeklarationer som görs inuti klassen
- *uppräknare* – värdena i **enum**-typer som deklareras inuti klassen fungerar som konstanta statiska datamedlemmar

En *datamedlem* eller *medlemsfunktion* kan vara

- *instansmedlem* – *icke*-**static** – varje objekt av klassen ifråga har en egen sådan
- *klassmedlem* – **static** – delas av alla objekt; ”tillhör” klassen

En *medlemsfunktion* kan vara

- *icke*-**const** – får ändra på datamedlemmars värde – *modifierare*
- **const** – får inte ändra på datamedlemmar värde – *accessor*
- *mycket* viktigt att deklarera en medlemsfunktion som inte ändrar som **const**
  - endast **const**-funktioner får användas på konstanta objekt
  - **const**- funktioner kan bara anropa andra **const**-funktioner
  - kompilatorn kontrollerar att **const** efterlevs
- det finns en kategori medlemsfunktioner som betecknas *speciella medlemsfunktioner*
  - mycket viktigt att förstå deras syfte och egenskaper
  - rör initiering, kopiering, tilldelning och destruering av objekt

Konstruktion av en enskild, icke-trivial klass

- övningsobjekt – en egen strängklass String
  - välkänd datatyp
  - containerliknande klass
- ett flertal problem ska lösas
  - intern representation – hur ska strängvärdet lagras? – representation för tom sträng?
  - initiering (konstruktorer)
  - destruering (destruktor)
  - kopiering – kopieringskonstruktor, kopieringstilldelningsoperator
  - flyttsemantik (move-semantik) – flyttkonstruktor, flyttilldelningsoperator
  - iteratorer †
  - operationer för övrigt
- intressanta frågor dyker upp
  - undantagssäkerhet
  - kodningstekniker
  - återanvändning
  - m.m.

† anger sådant som kan komma att hoppas över eller endast behandlas översiktligt under föreläsning – för självstudier!

Klassen String – funktionalitet

- initiering

```
String s1;                  // defaultinitiering, till tom sträng, ""
String s2{"foo"};           // med C-sträng (litteral eller variabel)
String s3{s2};              // genom kopiering av annan String
String s4{ 'A', 'h', 'a' }; // genom element från en initierarlista
```

- tilldelning

```
s1 = s2                     // String = String
s1 = "foo"                  // String = char*
s1 = { 'A', 'h', 'a' }      // String = initierarlista
```

- elementåtkomst

```
s1[i] = s2[j]               // utan indexkontroll
s1.at(i) = s2.at(j)         // med indexkontroll
```

- storlek

```
s1.length()                 // aktuell längd
s1.empty()                  // tom sträng?
s1.clear()                  // radera (gör till tomma strängen)
```

Klassen String – funktionalitet, forts.

- strängsammansättning

```
s1 + s2

s1 + "foo"          "foo" + s1

s1 += s2

s1 += "foo"
```

- likhet och olikhet

```
s1 == s2            s1 == "foo"         "foo" == s1

s1 != s2            s1 != "foo"         "foo" != s1
```

- jämförelser

```
s1 < s2         s1 <= s2        s1 > s2         s1 >= s2

s1 < "foo"      s1 <= "bar"     s1 > "foo"      s1 >= "bar"

"foo" < s       "bar" <= s      "foo" > s       "bar" >= s
```

Klassen String – funktionalitet, forts.

- byta innehåll med annat String-objekt

```
s1.swap(s2)          // som medlemsfunktion

swap(s1, s2)         // som normal funktion
```

- iteratorer †

```
for (auto it = s.begin(); it != s.end(); ++it)        // String::iterator
{
   cout << *it;
}

for (auto& c : s)                                     // char&
{
   c = tolower(c);
}
```

- in- och utmatning som för C-strängar

```
cout << s1;

cin >> s1;

getline(cin, s1, ';');          // '\n' är default om inget bryttecken anges
```

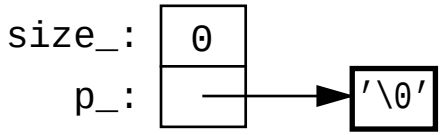
Klassen String (utvalda delar)

```
class String
{
public:
   using size_type = std::size_t;                  // nästlad typ

   String() = default;                             // defaultkonstruktor
   String(const String&);                          // kopieringskonstruktor
   String(String&&) noexcept;                      // flyttkonstruktor
   String(const char*);                            // typomvandlande konstruktor
   …
   ~String();                                      // destruktor

   String& operator=(const String&) &;             // kopieringstilldelningsoperator
   String& operator=(String&&) & noexcept;         // flytilldelningsoperator
   String& operator=(const char*) &;               // typomvandlande tilldelningsoperator
   …
   size_type length() const;                       // const-funktion – "accessor"
   bool      empty() const;
   void      clear();                              // icke-const-funktion – "modifierare"

   const char* c_str() const;                      // typomvandlande funktion

   void swap(String&);                             // bör alla containerliknande klasser ha
```

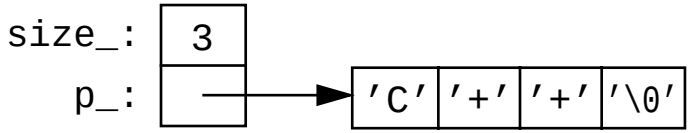
Klassen String, forts.

```
private:
   static char empty_rep_[1];                  // en statisk noll-avslutad C-sträng representerar tom sträng

   size_type size_{0};                         // aktuell storlek
   char* p_{empty_rep_};                       // pekare till teckenfält där strängvärdet lagras

   void construct_(const char*, size_type);    // hjälpfunktioner som används av konstruktorer
   …
   void append_(const char*);                  // hjälpfunktioner som används av tilldelningsoperatorer
   …
};
```

- alla datamedlemmar är private (*HIC++ 11.1.1*)
- alla String-objekt som är tomma strängar ska peka på gemensamma `empty_rep_`



- ett String-objekt som *inte* är en tom sträng har sin eget, dynamiskt allokerade minne för strängvärdet



Statisk datamedlem

```
static char empty_rep_[1];
```

- en statisk datamedlem ingår *inte* i objekten
  - *klassmedlem*
  - skapas och initieras i samband med att programmet startas
  - deklarationen i klassen är enbart en *deklaration*
- *definitionen* görs separat, i detta fall i filen String.cc

```
char String::empty_rep_[1];
```

  - **static** anges inte i definitionen
  - `String::` före namnet anger att `empty_rep_` är en medlem av `String`
  - det enda tecknet i fältet initieras automatiskt till `'\0'` – en tom C-sträng

Defaultkonstruktor

```
String() = default;                    // kompilatorn får generera!
```

- en defaultkonstruktor är en konstruktor som kan anropas *utan argument*

```
String s;                              // objektdeklaration utan initierare

String fun() { return String{}; }      // temporärt objekt skapas utan argument
```

  - följande konstruktor är både defaultkonstruktor och typomvandlande konstruktor

```
String(const char* cstr = "");

String s1;                             // använd som defaultkonstruktor

String s2{"C++"};                      // kan ta ett argument
```

- = **default** innebär att konstruktorn genereras av kompilatorn
  - medlemmar av enkel typ initieras endast om de har en initierare i sin deklaration
  - medlemmar av klasstyp initieras av någon konstruktor beroende på hur vi kodat, defaultkonstruktorn om inte annat
- de icke-statiska datamedlemmarna initieras av respektive *NSDMI* (”non-static data member initializer”)

```
size_type size_{ 0 };
char* p_{ empty_rep_ };
```

Medlemsinitierarlista

Förekommer inte i String men är en grundläggande konstruktion (exempel kommer lägre fram i kursen).

- en egen defaultkonstruktor hade kunnat skrivas

```
String()
  : size_{ 0 }, p_{ empty_rep_ }
{}
```

- en *medlemsinitierarlista* kan finnas mellan parameterlistan och funktionskroppen
  - inleds med *kolon* och består av kommaseparerade *medlemsinitierare*
  - skriv medlemsinitierarna i samma ordning som medlemmarna deklareras (*HIC++ 12.4.4*)
- om deklarationen av en datamedlem har en initierare utförs *inte* den om det finns en medlemsinitierare
  - ”dubbelinitiering” görs inte i sådana fall
  - specificera inte både NSDMI och medlemsinitierare i konstruktorer (*HIC++ 12.4.3*)
- samtliga konstruktorer för String visar sig med fördel kunna implementeras med hjälp av NSDMI

```
size_type size_{ 0 };
char* p_{ empty_rep_ };
```

  - inga medlemsinitierare förekommer i kodexemplet
  - därefter sker kompletterande åtgärder i de olika konstruktorerna med någon construct_()

Typomvandlande konstruktor

```
String(const char*);
```

- en konstruktor som kan anropas med *ett argument av en annan typ* är en typomvandlande konstruktor

```
String s{"C++"};
```

  - litteralen ”C++” har typen **const char**[4] – `s` har typen `String` – typomvandling sker
  - syntaxen kallas *direktinitiering* – kan även skrivas med vanliga parenteser

```
String s("C++");
```

- följande deklarationssyntax kallas *kopieringsinitiering*

```
String s1 = s2;                 // samma typ – kopieringskonstruktorn gör initiering

String s3 = "C++";              // implicit typomvandling – temporärt objekt skapas

String s4 = String{"C++"};      // explicit typomvandling på funktionsform – temporärt objekt skapas
```

  - flyttkonstruktorn gör initieringen i de två senare fallen (kommer strax…)
  - optimering kan förekomma
  - observera – denna syntax har inget med tilldelning att göra!

- alla explicita typomvandlingar sker med denna konstruktor, exempelvis

```
auto s5 = static_cast<String>("C++");
```

Destruktor

```
{
   String s{"foo"};                    // objektet s skapas
   …
   String* p = new String{"bar"};      // ett dynamiskt String-objekt skapas
   …
   delete p;                           // minnet för det dynamiska objektet återlämnas – objektet försvinner
   …
}                                      // deklarationsblocket för s lämnas – objektet s försvinner
```

- destruktorn körs alltid då ett String-objekt är på väg att försvinna
  - ska säkerställa att det dynamiska minnet återlämnas

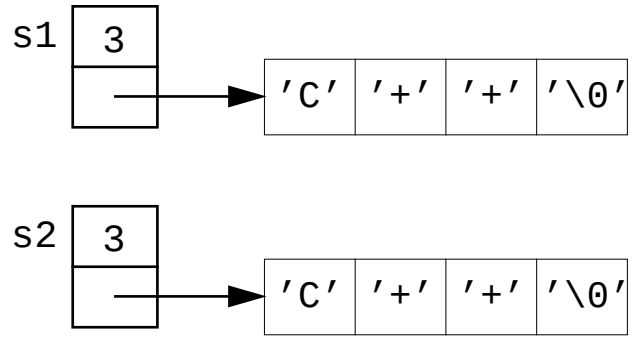
```
String::~String()
{
   if (!empty()) delete[] p_;
}
```

  - minnet har skapats av **new**[]
  - minnet måste återlämnas med **delete**[]
  - datamedlemmarna size_ och p_ försvinner med själva String-objektet
- kan något oönskat hända när **delete**[] utförs?

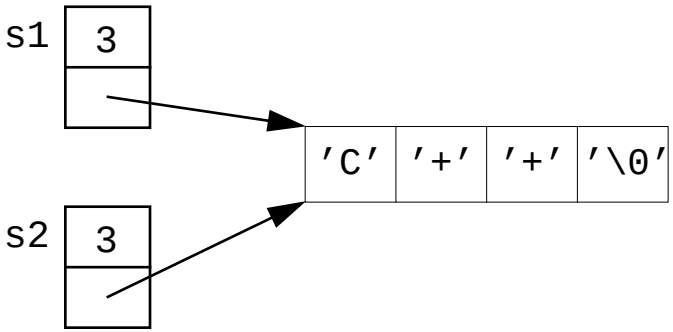
Kopieringskonstruktor

Djup kopiering.

```
String::String(const String& other)      // size_ och p_ initieras av sina initierare – tom sträng
{
   construct_(other.p_, other.size_);
}


String s1{"C++"};

String s2{s1};
```



- ett nytt objekt skapas – ingen historik som behöver tas hänsyn till
  - hjälpfunktionen construct_ tar hand om detaljerna
- den kompilatorgenererade konstruktorn skulle endast kopiera size_ och p_
  - teckenfältet skulle delas av flera objekt



Hjälpfunktioner för konstruktion och tilldelning †

Privata hjälpfunktioner används för att utföra detaljerna vid initiering och tilldelning. Exempel:

```
void String::construct_(const char* cstr, size_type size)
{
    if (cstr != nullptr && size > 0)
    {
        p_ = strcpy(new char[size + 1], cstr);
        size_ = size;
    }
}
```

- ”farliga saker” görs på speciella ställen – i construct_() och append_()
  - dynamiska minnesallokering – om **new** misslyckas kastas undantaget bad_alloc
  - inget kan hända vid kopieringen av tecknen
  - inget kan hända vid tilldelningen av size_
- om **new** kastar avbryts construct_() och i sin tur konstruktorn (alternativt tilldelningsoperatorn i fråga)
  - objektet kunde inte skapas (tilldelas)
  - det är inget som behöver göras på grund av att undantag kastas
- implementering av construct_() ovan hanterar de tre tänkbara fallen
  - cstr är en tompekare (**nullptr**) – borde inte ske men är möjligt – resultatet blir en tom sträng
  - cstr pekar på en tom sträng (size == 0) – resultatet ska vara en tom sträng
  - cstr pekar på en icke-tom sträng – resultatet ska vara en kopia av indata

Undantagssäkerhet †

Rimligt beteende om undantag kastas.

Tre nivåer:

- *grundläggande garanti – undantagssäkert*
  - om undantag kastas förblir objekt i ett tillåtet tillstånd – kanske inte det ursprungliga dock
  - inga resurser förloras – till exempel dynamiskt minne
- *stark garanti – starkt undantagssäkert*
  - operation lyckas antingen helt, ***eller***
  - så kastas undantag ***men***
  - programmet bibehåller det tillstånd det hade innan operationen påbörjades – inga objekt påverkas
- *kastar-inte-garanti*
  - operationen kastar inte undantag
  - destruktorer kastar aldrig undantag – deklarera dock *inte* det med **nothrow**

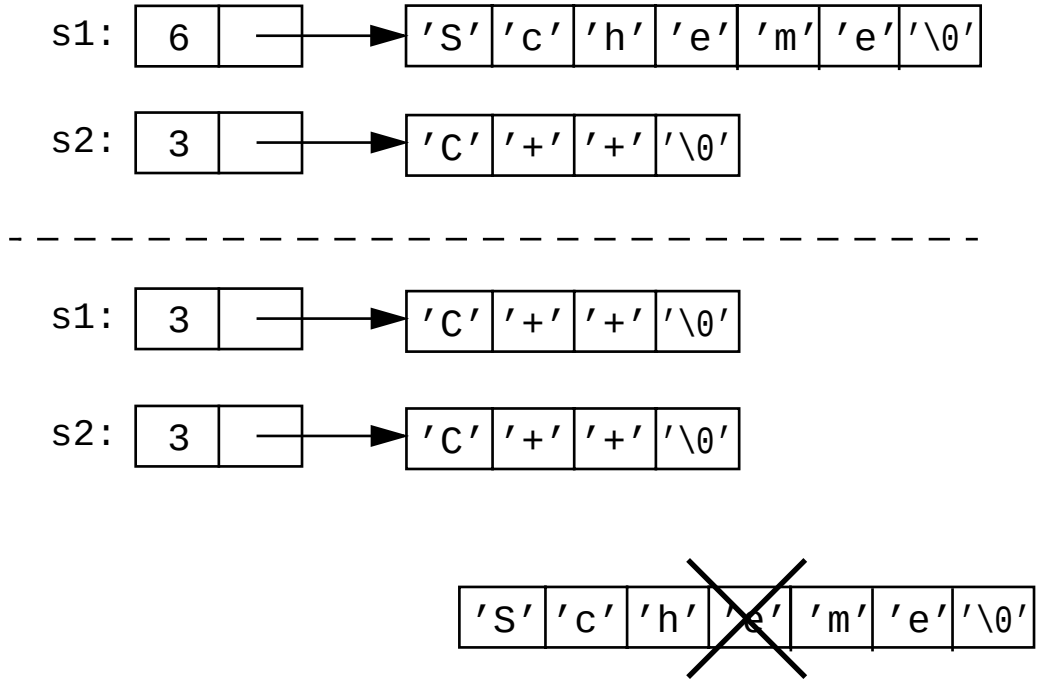
Undantagsneutralitet

- en undantagsneutral funktion vidarebefordrar kastade undantag
  - normalt samma undantag som ursprungligen kastats
  - lokala åtgärder kan ha behövt vidtas innan undantaget vidarebefordras

Kopieringstilldelningsoperator

Djup tilldelning.

```
s1 = s2;
```



- vänsteroperanden har historik
  - gammalt innehåll behöver tas om hand
  - kopiera nytt värde från högeroperanden
- viktigt att göra saker i rätt ordning
  - inget objekt ska hamna i ett odefinierat tillstånd
  - se först till att nytt minne erhålls
  - gör sedan ändringar
- glöm inte möjligheten

```
s1 = s1;
```


  - inte alltid så uppenbart…
- den kompilatorgenererade kopieringstilldelningsoperator skulle endast ha tilldelat size_ och p_
  - s1 och s2 hade kommit att dela på samma teckenfält
  - det gamla teckenfältet för s1 hade tappats bort (minnesläcka)
  - den som först försvinner tar bort värdet för den andra

Kopieringstilldelningsoperator – rättfram implementering

```
String& String::operator=(const String& rhs) &              // ref-qualifier, HIC++ 12.5.7
{
    if (this != &rhs)
    {
        char* p{empty_rep_};                                // i fall rhs är en tom String
        if (!rhs.empty())
            p = strcpy(new char[rhs.size_ + 1], rhs.p_);     // om vi överlever detta är vi säkra
        size_ = rhs.size_;
        if (!empty()) delete[] p_;
        p_ = p;
    }
    return *this;
}
```

- vänsteroperanden har gammalt innehåll att ta hand om
- kontrollerar om vänster och höger operand är samma objekt – självtest
  - **this** är en pekare till det objekt som medlemsfunktionen har anropats för – i detta fall en pekare till vänsteroperanden

    ```
    s = s
    ```

- utför djup kopiering på ett starkt undantagssäkert sätt – allokerar minne innan något ändras – om **new** kastar
  - inget minne kommer att läcka
  - inget av objekten kommer att vara ändrade
- egenskaper som en inbyggd tilldelningsoperator – returnerar icke-const referens (*lvalue*) till vänsterargumentet, String&

Kopieringstilldelningsoperator – elegant implementering

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

- använder idiomet *skapa en temporär och byt* (“create a temporary and swap”, *HIC++ 12.5.6*)
- *ref-qualifier* & medför att **operator**= endast kan användas om vänsteroperanden är ett *lvalue*-uttryck (namn på variabel)
  - typen för vänsteroperanden är *lvalue-referens till String* (String&)
  - tilldelningsoperatorer ska alltid deklareras så – får samma egenskap som de inbyggda operatorerna har
- behöver en funktion som kan byta innehåll hos två String-objekt på ett säker sätt, helst en *nothrow swap* (**noexcept**)
- en temporär skapas och initieras av kopieringskonstruktorn – djup kopia av rhs
- innehållet hos vänsteroperanden (***this**) och temporären byts – ***this** blir en kopia av `rhs`
  - temporären tar över det gamla innehållet hos ***this** – speciellt det dynamiskt allokerade teckenfältet
  - temporären destrueras efter swap genomförts – det gamla dynamiska minnet för ***this** återlämnas
- om ett undantag kastas, kommer det att inträffa då temporären initieras av kopieringskonstruktorn
  - *starkt undantagssäkert* – inget minne läcker – inga objekt hamnar i ett odefinierat tillstånd
  - *undantagsneutralt* – ett undantag som kastas förs vidare som det är
- självtest skulle kunna göras

```
if (this != &rhs) String{ rhs }.swap(*this);
```

De användbara swap-funktionerna

Grundregel: alla datatyper som kan ha swap() bör ha det.

- swap() som medlem – anropar standardbibliotekets swap

```
void String::swap(String& rhs) noexcept
{
   std::swap(p_, rhs.p_);
   std::swap(size_, rhs.size_);
}
```

- swap() som normal funktion – anropar medlemmen

```
void swap(String& lhs, String& rhs) noexcept
{
   lhs.swap(rhs);
}
```

  - två viktiga aspekter – återanvändning samt undvika att göra till **friend**
  - specialisering av swap() för String – väljs i stället för standardbibliotekets swap() då argumenten är String-objekt

- std::swap() kastar inget undantag om inte **operator**= för typen ifråga kastar undantag
  - **operator**= för grundläggande typer och pekare kastar inte undantag
  - String::swap() kastar inte undantag

```
void String::swap(String& rhs) noexcept;

void swap(String& lhs, String& rhs) noexcept;
```

Typomvandlande tilldelningsoperator †

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

s1 = "foobar";
```

- typomvandlande tilldelningsoperator från C-sträng (**char**[], **char***) till String
- implementeras med idiomet *skapa en temporär och byt*
  - konstruktorn String(**const char***) används för att skapa det temporära objektet

Typomvandling †

- Vi har sett två exempel redan

```
String(const char* rhs);                    // typomvandlande konstruktor
String& operator=(const char* rhs);         // typomvandlande tilldelningsoperator
```

  - en sådan konstruktor kan användas implicit

```
String s = "C++";                           // implicit typomvandling, temporär skapas
```

  - vill man förbjuda det kan man deklarera konstruktorn **explicit** (vill vi inte för String)

```
explicit String(const char* rhs);
```

- det finns en *vanlig medlemsfunktion* som gör typomvandling från String till **const char*** (C-sträng)

```
const char* c_str() const { return p_; }
```

- man kan deklarera en *typomvandlande operator* (medlem) som gör typomvandling från String till **const char***

```
operator const char*() const { return p_; }
const char* p{s};                           // implicit typomvandling
```

  - deklarera inte implicit typomvandling – deklarera **explicit** (*HIC++ 12.1.1*)

```
explicit operator const char*() const { return p_; }
const char* p = static_cast<const char*>(s);   // explicit typomvandling
```

Flyttsemantik

En av de stora nyheterna i C++11 – alternativ till traditionell *kopieringssemantik*.

- *temporära objekt* skapas i många olika situationer – ofta implicit
- om ett temporärt objekt används för att initiera eller tilldela ett annat objekt är det onödigt att göra en kopia
  - kopiering kan vara kostsamt – tid och utrymme
  - flytta i stället innehållet från temporären till destinationsobjektet
- hur finner man sådana objekt – de syns ju vanligvis inte?
  - kompilatorn vet!
  - *rvalue-referenser* fångar dem automatiskt!
  - vi behöver bara vara medvetna om möjligheten och ha det i åtanke när vi konstruerar klasser

```
String(const String&);                  // denna kan fånga alla slags objekt
String(String&&) noexcept;              // men denna är en bättre match för temporära objekt (rvalue)

String& operator=(const String&) &;
String& operator=(String&&) & noexcept;
```

- implementering av flyttsemantik
  - ett objekt vars resurser har flyttats måste vara *destruerbart*
  - ibland vill vi applicera flyttsemantik även på vanliga objekt (variabler)
  - ett objekt vars innehåll har flyttats måste vara *tilldelningsbart* och *kopierbart*
  - bör motsvara ett defaultinitierat objekt – tom sträng i fallet String
- flyttkonstruktor och flyttilldelningsoperator ska alltid deklareras **noexcept** – inga undantag ska kastas

Hjälpfunktionen std::move()

Det traditionella sättet att byta värde på två String-variabler:

```
void swap(String& x, String& y)
{
   String tmp{ x };              // x kopieras till tmp av kopieringskonstruktorn
   x = y;                        // y kopieras till x av kopieringstilldelningsoperatorn
   y = tmp;                      // tmp kopieras till y av kopieringstilldelningsoperatorn
}
```

Både x och y ska erhålla nya värden – deras gamla värden behöver inte behållas då de kopieras – *flytta i stället*

```
#include <utility>

void swap(String& x, String& y)
{
   String tmp{ std::move(x) };   // x:s värde flyttas till tmp av flyttkonstruktorn
   x = std::move(y);             // y:s värde flyttas till x av flyttilldelningsoperatorn
   y = std::move(tmp);           // tmp:s värde flyttas till y av flyttilldelningsoperatorn
}
```

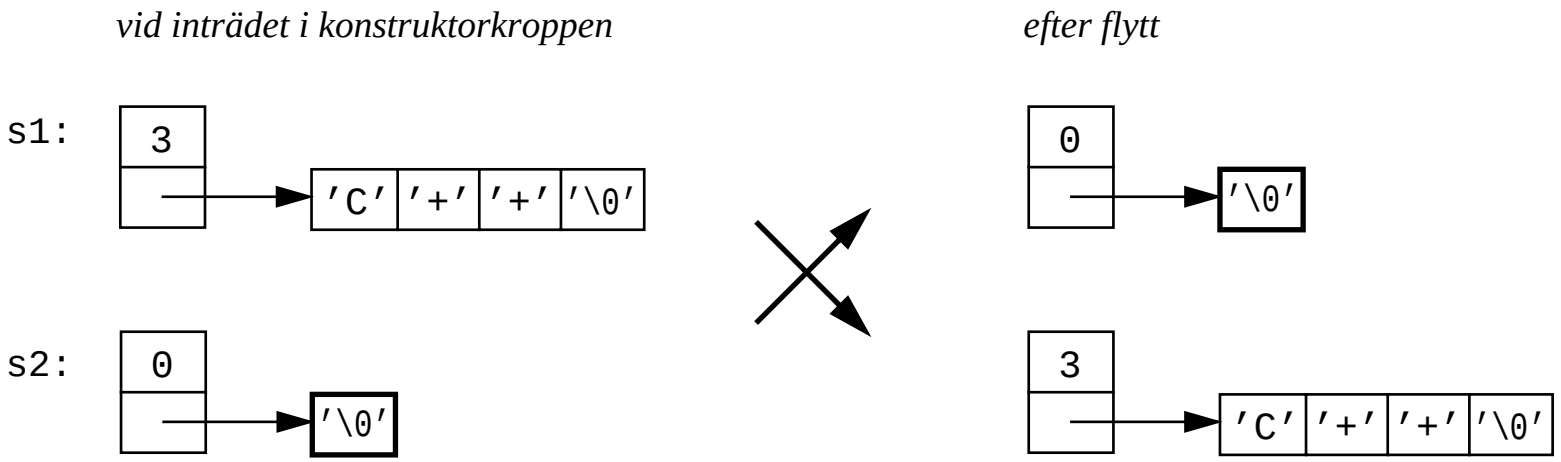
- hjälpfunktionen move() gör inget mer än att typomvandlar sitt argument till en *rvalue-referens* – `String&&` i detta fall
- detta triggar användning av flyttoperationerna

Anm. move() gör i grunden typomvandlingen **`static_cast`**`<String&&>`

Flyttkonstruktor

```
String::String(String&& other) noexcept      // size_ och p_ initieras av sina NSDMI – tom sträng initialt
{
    swap(other);                             // byt innehåll med other – other blir tom sträng
}

String s1 = String{"C++"};

String s2{ std::move(s1) };      // hjälpfunktionen move() omvandlar s1 (lvalue) till String&& (rvalue-referens)
```

- flyttkonstruktorn väljs i exemplet ovan, om den finns, i stället för kopieringskonstruktorn
- initieringen av s2 innebär följande

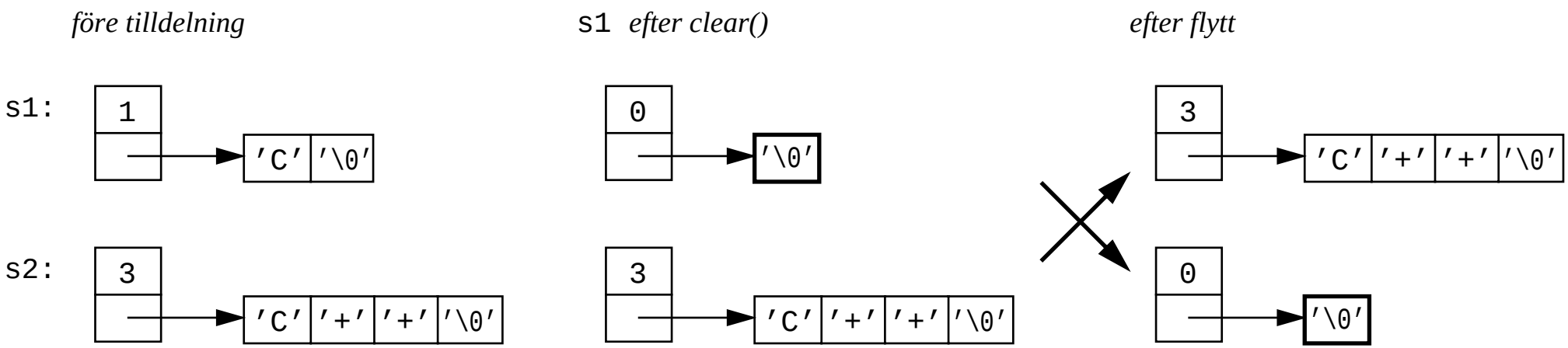


- bör deklareras **noexcept** (*HIC++ 12.5.4*)

Flyttilldelningsoperator

Vår implementering använder clear() och swap():

```
String& String::operator=(String&& rhs) & noexcept          // HIC++ 12.5.7, 12.5.4
{
    clear();                // vänsteroperanden sätts till ”tom sträng”
    swap(rhs);              // flytt utförs genom att byta innehåll på vänster- och högeroperanden
    return *this;
}

s1 = std::move(s2);
```



Ett alternativ är att bara låta objekten byta innehåll, dvs utan att först göra clear() på vänsterargumentet.

Regler för generering av flyttkonstruktor och flyttilldelningsoperator †

Flyttkonstruktor genereras endast om klassen

- *inte* har en användardeklarerad kopieringskonstruktor
- *inte* har en användardeklarerad kopieringstilldelningsoperator
- *inte* har en användardeklarerad flyttilldelningsoperator
- *inte* har en användardeklarerad destruktor

Flyttilldelningsoperator genereras endast om klassen

- *inte* har en användardeklarerad kopieringskonstruktor
- *inte* har en användardeklarerad flyttkonstruktor
- *inte* har en användardeklarerad kopieringstilldelningsoperator
- *inte* har en användardeklarerad destruktor

Konstruktor som tar en initierarlista †

```
String::String(std::initializer_list<char> il)
{
    construct_(il);
}

void String::construct_(initializer_list<char> il)
{
    if (il.size() > 0)
    {
        size_ = il.size();
        p_ = new char[size_ + 1];
        std::copy(il.begin(), il.end(), p_);
        p_[size_] = '\0';
    }
}

String s{ 'G', 'a', 'z', 'o', 'n', 'k' };
```

- `std::initializer_list` är en klassmall som instansieras för elementtypen ifråga (**char**)
- `il` initieras från initierarlistan som anges i deklarationen av String-objektet
- `std::initializer_list` har iteratorer och tre operationer – size(), begin() och end()

Tilldelningsoperator som tar en initierarlista †

```
String& String::operator=(initializer_list<char> rhs) &
{
    String{rhs}.swap(*this);
    return *this;
}

s = { 'G', 'a', 'z', 'o', 'n', 'k' };
```

- implementeras med idiomet ”skapa en temporär och byt”
- konstruktorn som tar en initierarlista används för att skapa ett temporärt objekt

Delegerande konstruktorer

Inga konstruktorer hos String lämpar sig för att exemplifiera detta.

```
class List
{
public:
   List() : head_{ new Null_Node; } {}                  // en tom lista innehåller en Null_Node

   List(List&& other) noexcept : List() { swap(other); }

   void swap(List& other) noexcept { std::swap(head_, other.head_); }

   ...

private:
   List_Node* head_;
};
```

- flyttkonstruktorn *delegerar* till defaultkonstruktorn att initiera det nya `List`-objektet till ”tom lista”
- sedan byts innehåll med `other`
- *men* vi vill egentligen **inte** hantera Null_Node med dynamisk minnestilldelning – annan implementering för ”tom lista” bör användas!

Kompilatorgenererade versioner av speciella medlemsfunktioner †

- *defaultkonstruktorn* skulle motsvara

```
String() {}
```

  - datamedlemmar av *grundläggande typ*, t.ex. **int**, eller *pekare* initieras *inte*
  - datamedlemmar av *klasstyp* initieras av sin *defaultkonstruktor*
  - *om* det finns initierare i deklarationerna av datamedlemmar utförs de

- *kopieringskonstruktorn* skulle *kopieringsinitiera* datamedlemmarna i den ordning de deklareras

```
String(const String& other)
  :    size_{ other.size_ }, p_{ other.p_ }
{}
```

- *flyttkonstruktorn* skulle *flyttinitiera* datamedlemmarna i den ordning de deklareras

```
String(String&& other)
  : size_{ std::move(other.size_) }, p_{ std::move(other.p_) }
{}
```

  - ingen egentlig skillnad jämfört med kopieringskonstruktorn för dessa typer

- *destruktorn* skulle motsvara

```
~String() {}
```

  - datamedlemmar av *klasstyp* destrueras av sina destruktorer (i omvänd deklarationsordning)
  - **delete**[] p_ görs inte – dynamiska minnet skulle läcka

Kompilatorgenererade versioner av speciella medlemsfunktioner, forts. †

- *kopieringstilldelningsoperatorn* skulle *kopieringstilldela* datamedlemmarna i den ordning de deklareras

```
String& operator=(const String& rhs) &
{
    size_ = rhs.size_;
    p_    = rhs.p_;
}
```

  - kopieringstilldelningsoperatorn för respektive typ utförs

- *flyttilldelningsoperatorn* skulle *flyttilldela* datamedlemmarna i den ordning de deklareras

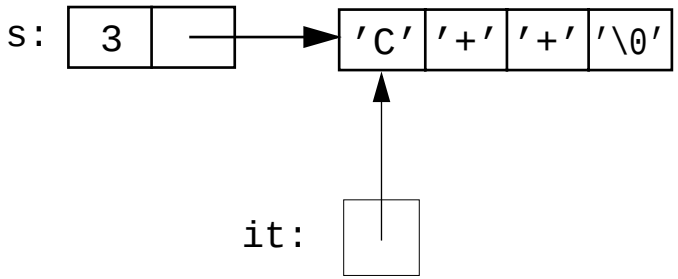
```
String& operator=(String&& rhs) & noexcept
{
    size_ = std::move(rhs.size_);
    p_    = std::move(rhs.p_);
}
```

  - flyttilldelningsoperatorn för respektive typ utförs
  - size_ och p_ är typer för vilka inte flyttilldelning finns (är meningsfull)

Vi måste alltså deklarera samtliga dessa själva för att erhålla korrekt initiering, destruering och kopiering – defaultkonstruktorn kunde dock genom NSDMI kompilatorn få generera.

Iteratorer för String †

String är en containertyp med element av typen **char**.




```
class String
{
public:
    …
    using iterator               = char*;
    using const_iterator         = const char*;

    using reverse_iterator       = std::reverse_iterator<iterator>;
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;
    …
```

- förutsättningarna för Random Access-iteratorer uppfylls
- iteratorer kan definieras så enkelt som teckenpekare
- bakåtiteratorer definieras med hjälp av mallen `std::reverse_iterator`

Iteratorer för String, forts. †

- String ska ha full uppsättning iterator-funktioner

```
iterator       begin() { return iterator{p_}; }
const_iterator begin() const { return const_iterator{p_}; }
iterator       end() { return iterator{p_ + size_}; }
const_iterator end() const { return const_iterator{p_ + size_}; }

reverse_iterator       rbegin() { return reverse_iterator{end()}; }
const_reverse_iterator rbegin() const { return reverse_iterator{end()}; }
reverse_iterator       rend() { return iterator{begin()}; }
const_reverse_iterator rend() const { return const_iterator{begin()}; }

const_iterator         cbegin() const { return const_iterator{p_}; }
const_iterator         cend() const { return const_iterator{p_ + size_}; }
const_reverse_iterator crbegin() const { return const_reverse_iterator{end()}; }
const_reverse_iterator crend() const { return const_reverse_iterator{begin()}; }
```

- eftersom iteratorerna är vanliga pekare fungerar de inbygga operatorerna för stegning, jämförelse, avreferering, etc.

```
for (String::const_iterator it = s.cbegin(); it != s.cend(); ++it)
{
    cout << *it;
}
```

  - men hellre **auto** för att deklarera `it` – ännu hellre intervallstyrd **for**-sats om det passar
  - ”range access”-funktionerna kan användas

Namnrymd för String

- namnrymden `IDA_String` introduceras i inkluderingsfilen (*orginal namespace definition*):

```
#ifndef STRING_H
#define STRING_H

namespace IDA_String
{
    // Klassdefinitionen och deklarationer för tillhörande icke-medlemsfunktioner
}

#endif
```

- och utvidgas i implementeringsfilen (*extension namespace definition*) :

```
#include "String.h"

namespace IDA_String
{
    // Separata definitioner för medlems- och icke-medlemsfunktioner
}
```

- det kan vara nödvändigt att deklarera en klass och dess icke-medlemsfunktioner i samma namnrymd för att kompilatorn/länkaren ska hitta funktionerna.