EDAF30 – Programmering i C++

1. Introduktion

Sven Gestegård Robertz
Datavetenskap, LTH

2017

Innehåll

EDAf30: Programmering i C++, 7.5 hp

Kursens syfte är att ge kunskaper i objektorienterad programmering i C++.

Mål:

- Kunskap och förståelse
  - känna till och kunna beskriva skillnaderna mellan C++ och Java
  - vara förtrogen med språket C++ och standardbiblioteket STL
  - kunna förklara grundläggande begrepp inom objektorienterad C++-programmering
  - kunna tolka, analysera och förklara befintlig C++-kod.
- Färdighet och förmåga
  - kunna utveckla ett fungerande C++-program från en given specifikation
  - kunna felsöka metodiskt i C++-kod.

EDAf30: Programmering i C++, 7.5 hp
Viktiga skillnader mot Java

Nya eller utökade begrepp i C++
(jämfört med Java / grundkursen):

- Pekare och minneshantering
- Funktioner: värde- och referensanrop
- Polymorfism: både statisk och dynamisk
  (Jfr *templates* med *generics*)

EDAf30: Programmering i C++, 7.5 hp
Obligatoriska moment

Kursen examineras genom

- laborationer
- inlämningsuppgifter
- skriftlig tentamen

Slutbetyget baseras på den skriftliga tentamen.

EDAf30: Programmering i C++, 7.5 hp
Administration

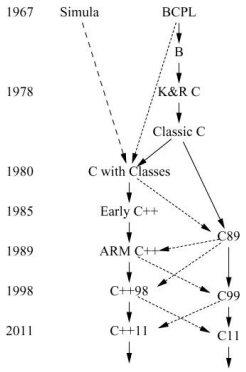
- Kursprogram
- Kurregistrering
- Anmälan till laborationer
  - Görs på webben, länk på kurshemsidan
  - Anmälan görs till en labbgrupp – samma tid alla veckor

Historik
C++ härstammar från Simula och C.

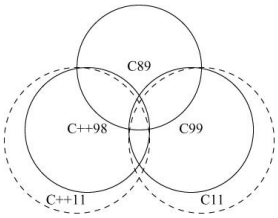
1967: Simula (Dahl & Nygaard)
1972: C (Dennis Ritchie)
1978: K&R C (Kernighan & Ritchie)
1980: C with Classes (Bjarne Stroustrup)
1985: C++ (Bjarne Stroustrup)
- ISO-standard 1998

Andra släktingar:

1995: Java (James Gosling et al.)
2000: C# (Anders Hejlsberg)
- virtuell maskin
- automatisk minneshantering
- *säkra* språk



C++ är inte en ren utökning av C



- både ISO C och ISO C++ härstammar från K&R C, och är "syskon"
- Vissa detaljer är inkompatibla mellan C och C++
- Ytorna är inte skalenliga

Generellt: skriv inte C++ som om det vore C

Vad är C++?

ISO-standarden för C++ definierar två saker
- *Språket* C++ (*Core language features*), t ex
  - datatyper (t ex **`char`**, **`int`**)
  - kontrollflödesmekanismer (t ex **`if`**- och **`while`**-satser).
- *Standardbiblioteket* (*Standard-library components*), t ex
  - Datastrukturer (t ex `string`, `vector` och `map`)
  - Operationer för in- och utmatning (t ex << och `getline()`)
  - Algoritmer (tex `find()` och `sort()`)
  - Deklarationerna *inkluderas* med t ex **`#include`** <vector>
- Standardbiblioteket är skrivet i vanlig C++
  - Exempel på vad man kan göra

Ett minimalt program i C++

empty.cc

```
int main( ) { }
```

- har inga parametrar
- gör ingenting
- returvärdet från `main()` tolkas av systemet som en felkod
  - skilt från noll betyder fel
  - inget returvärde tolkas som noll (OBS! endast i `main()`)
  - används sällan i Windows
  - används ofta på Linux/Mac

Ett första program i C++
Hello, World!

hello.cc

```
#include <iostream>
int main( )
{
   std::cout << "Hello, World!" << std::endl;
   return 0;
}
```

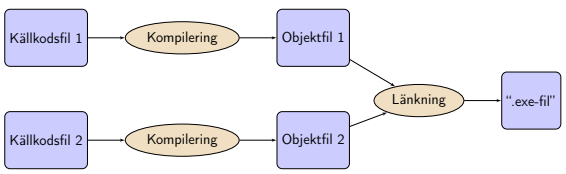
hello.cc

```
#include <iostream>
using std::cout;
using std::endl;

int main( )
{
   cout << "Hello, World!" << endl;
   return 0;
}
```


Vad är ett program?

C++ är ett kompilerat språk
- Källkod
- Objektfil
- Exekverbar fil



Ett program i C++

Exempel: beräkna och skriv ut x^2.

```
#include <iostream>

double square(double x)
{
  return x*x;
}

void print_square(double d)
{
    std::cout << "the square of " << d <<
                 " is " << square(d) << std::endl;
}

int main( )
{
    print_square(1.234);
    return 0;
}
```


Funktioner
Deklaration och definition

Det huvudsakliga sättet att få något gjort i C++:

- Anropa en funktion som gör det
  - En funktion måste ha blivit *deklarerad* för att kunna anropas
  - En funktionsdeklaration anger
    - namn
    - returtyp
    - typer för argument (parametrar)
  - Exempel

```
int random();
void exit(int);
double square(double);
int pow(int x, int exponent);
```

    - Kompilatorn ignorerar parameternamn
    - Ange namn om det ökar läsbarhet
- En *definition* av en funktion innehåller implementationen
  - Får bara finnas på ett ställe

Skillnad från Java

- I Java kan bara funktioner och variabler deklareras inuti klasser.
- I C++ kan funktioner och variabler finnas oberoende av klasser.
  - fria funktioner: tillhör inte någon klass
  - medlemsfunktioner i en klass
  - globala variabler
  - medlemsvariabler

Funktionsdefinition
Exempel

- Deklaration och definition

Exempel: Medelvärde – variant 1

```
double medelv(double x1, double x2) // Deklaration och definition
{
   return (x1+x2)/2;
}

int main()
{
   double a=2.3, b=3.9;
   cout << medelv(a, b) << endl;
}
```


Funktionsdefinition
Med tidigare deklaration

- "Framåtdeklaration" (*Forward declaration*)
- Funktionsdefinition efter main

Exempel: Medelvärde – variant 2

```
double medelv(double, double);        // deklaration (prototyp)

int main()
{
   double a=2.3, b=3.9;
   cout << medelv(a, b) << endl;      // användning
}

double medelv(double x1, double x2)   //definition
{
   return (x1+x2)/2;
}
```


Funktionsdefinition
Med tidigare deklaration

- "Framåtdeklaration" (*Forward declaration*)
- Funktionsdefinition efter main

Exempel: Medelvärde – variant 2

```
double medelv(double, double);        // deklaration (prototyp)

#include "medelv.h"

int main()
{
   double a=2.3, b=3.9;
   cout << medelv(a, b) << endl;      // användning
}

double medelv(double x1, double x2)   //definition
{
   return (x1+x2)/2;
}
```

Uppdelning av program i flera filer

- Hantera deklarationer och definitioner
  - Definitionen av varje funktion får bara finnas på ett ställe
  - Deklarationen behövs överallt där funktionen används
  - `#include`

Uppdelning av program i flera filer
Exempel: Headerfil

Minimalt exempel:
medelv – ett bibliotek för att beräkna medelvärde

- medelv.h
- medelv.cc

Användning:

- main.cc

medelv.h: deklarationer

```
double medelv(double x1, double x2);
double medelv(int x1, int x2);
```


Uppdelning av program i flera filer
Exempel: Källfil

medelv.cc: definitioner

```
#include "medelv.h"    // gör deklarationerna synliga
                       // så att kompilatorn kan kontrollera dem
double medelv(double x1, double x2)
{
   return (x1+x2)/2;
}

double medelv(int x1, int x2)
{
   return static_cast<double>(x1 + x2) / 2;
}
```


Uppdelning av program i flera filer
Exempel: Huvudprogram

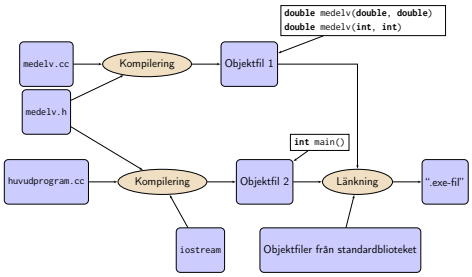
main.cc: användning

```
#include <iostream>
using std::cout;
using std::endl;

#include "medelv.h"     // gör deklarationer synliga
                        // för att kunna använda dem
int main()
{
   double a=2.3;
   double b=3.9;
   int m=3;
   int n=4;
   cout << medelv(a, b) << endl;
   cout << medelv(m, n) << endl;
}
```


Uppdelning av program i flera filer

- Placering av funktionsdeklarationer i s.k. headerfiler (.h)
- Uppdelning av koden på flera källfiler (.cc)
- Separatkompilering av varje källfil (.cc)
- Länkning av program



Funktioner
Anrop

- Vid funktionsanrop *kopieras värdena* för argumenten
  - Typkontroll
  - Implicit typomvandling
- Exempel: med en funktion `double square(double)`

```
double s2 = square(2);       // 2 konverteras till double
double s3 = square("three"); // fel
```

- Overloading ("överlagring")
  - Får inte skilja enbart i returtyp
  - Måste vara entydigt

```
void print(int);
void print(double);
void print(std::string);

void user()
{
    print(42);   // anropar print(int);
    print(1.23); // anropar print(double);
    print(4.5f); // anropar print(double);
    print("Hej") // anropar print(std::string);
}
```

Funktioner
Anrop - entydighet

- Vid funktionsöverlagring väljer kompilatorn "den bästa" funktionen (efter implicit typkonvertering)
- Om två alternativ är "lika bra" fås ett kompileringsfel

```
void print2(int, double);
void print2(double, int);

void user()
{
    print2(0, 0);   // Fel! inte entydigt
}
```

- och även (med `print()` från förra bilden)

```
    long l = 17;
    print(l);        // Fel! print(int) eller print(double)?
```


Funktioner
Tumregel

- En funktion ska bara ha en uppgift
- Håll funktioner korta, om möjligt
- Tumregel
  - Max 24 rader
  - Max 80 tecken per rad
  - Max 3 indenteringsnivåer
  - Max 5–10 lokala variabler
  - Omvänt proportionellt mot komplexitet

Värdeanrop och referensanrop
Värdeanrop (*call by value*)

Vid 'vanliga' anrop kopieras alltid värdena från de aktuella parametrarna till de formella (vilka fungerar som lokala variabler)

Exempel: Byta plats på två heltalsvärden

```
void swap(int a, int b)
{
   int tmp=a;
   a = b;
   b = tmp;
}

...  och användning:

int x = 2;
int y = 10;
...
swap(x, y);
```

Innehållet i x och y ändras inte

Värdeanrop och referensanrop
Referensanrop (*call by reference*)

Istället för *värdeanrop* används *referensanrop*:

Exempel: Byta plats på två heltalsvärden

```
void swap(int& a, int& b)
{
   int tmp=a;
   a = b;
   b = tmp;
}
```

NB! Värdet på en referens-parameter måste vara ett *lvalue*

Anropet `swap(x,15);` ger felmeddelandet

```
invalid initialization of non-const reference of type 'int&'
from an rvalue of type 'int'
```


Referenser

- En referens är *ett alias* till en variabel

Datatyper och variabler

- Varje namn och varje uttryck (*expression*) har en typ
- Några begrepp:
  - en *deklaration* introducerar ett *namn*
  - en *typ* definierar mängden möjliga värden och operationer (för ett *objekt*)
  - ett *objekt* är ett stycke minne som innehåller ett *värde*
  - ett *värde* är en följd bitar som ska tolkas enligt en viss *typ*.
  - en *variabel* är ett namngivet *objekt*

Datatyper
Primitiva typer

- Heltalstyper: **char**, **short**, **int**, **long**, **long long**
  - **signed** (som i Java)
  - **unsigned** (*modulo* 2^N "icke-negativa" tal, finns inte i Java)
- Flyttalstyper: **float**, **double**, **long double**
- **bool** (boolean i Java)

- Typen **char** är "den naturliga storleken för ett tecken" på en given maskin (oftast 8 bitar). Storleken kallas (i C/C++) "en byte" oavsett antal bitar.
- **sizeof**(**char**) $\equiv 1$
- Alla andra datatypers storlek är multipler av **sizeof**(**char**).
  - Storlekar är *implementation defined*
  - **sizeof**(**int**) är ofta 4.

Operatorer

Operatorer och uttryck liknar Java

Gemensamma med Java

T ex `+ - * / % ++ -- += -= *= && || & |` etc. samt `[] . ?:`

Trinära villkorsoperatorn ?: (som i Java)

```
z = (x>y) ? x : y;
```

```
if (x>y)
  z=x;
else
  z=y;
```

Många fler, t ex

Pekaroperatorer: `* & ->`
In- och utmatning: `<< >>`
sizeof, **decltype**

Variabler
Deklaration och initiering

Deklaration utan initiering (undvik)

```
int x;        // x har odefinierat värde
              // (jfr lokal variabel i Java)
```

Deklaration och initiering

```
int x{7};     // Rekommenderad stil i C++
int y = {7};  // C++ med extra =
int z = 7;    // C-stil
```

C-stil: Se upp med implicit typkonvertering

```
int x = 7.8;  // x == 7. Ger ingen varning
int y {7.8};  // ger varning (eller fel med -pedantic-errors)
```


Variabler
Automatisk typinferens

auto: Man behöver inte ange typen på en variabel om den kan härledas från initieringen.

Deklaration och initiering

```
auto x = 7;      // int x
auto c = 'c';    // char c
auto b = true;   // bool b
auto d = 7.8;    // double d
```

NB! med **auto** finns ingen risk för felaktig typkonvertering, så det är säkert att använda tilldelning (=).

Använd inte **auto** om du behöver vara explicit med typdeklarationen, t ex om

- att ange typen gör koden mer lättläst
- man vill bestämma talområde eller precision (t ex **int**/ **long** eller **float**/ **double**)

Datatyper
Pekare, Arrayer och Referenser

- Referenser (konstanta och kan inte vara "null")[1]
- Pekare
  - addressen till ett objekt
  - heltalstyp, aritmetik, till skillnad från i Java
- Arrayer ("vektorer"). Liknar Java, fast
  - Osäkra
    - En array vet inte hur stor den är.
  - En array kan ha godtycklig typ som element
    - inte bara primitiva typer eller referenser som i Java
  - C-strängar är **char**[] som är *null-terminerade*.
    Exempel: **char** s[6] = "Hello";

s:	'H'	'e'	'l'	'l'	'o'	'\0'

(*Mer om detta senare*)

[1] Egentligen inte en typ, utan ett *alias* till ett objekt

Datatyper
Användardefinierade typer

- **struct**, **class** (som **class** i Java)
- Exempel från standardbiblioteket
  - std::string (liknar java.lang.String)
  - std::vector, std::list ... (liknar motsvarande i java.util)
- **enum class**: uppräkningstyp (liknar **enum** i Java)

Typdeklarationer med `typedef` och `using`

`typedef`

```
typedef unsigned long ulong;
typedef std::vector<int> IntVector;
```

`using` (C++ 11)

```
using ulong = unsigned long;
using IntVector = std::vector<int>;
```

Användning

Följande deklarationer är ekvivalenta:

```
ulong a[12];     unsigned long a[12];
IntVector vec;   std::vector<int> vec;
```


Vi har talat om

- Kursen
- C++
  - historik
  - funktioner
  - datatyper och variabler
  - deklarationer och definitioner
  - jämförelser med Java och C

Nästa föreläsning: *User-defined types*

- klasser
- uppräkningstyper
- operatorer
- Introduktion till I/O

Läsanvisningar

Referenser till relaterade avsnitt i Lippman

Funktioner	6.1 (s 201–207)
Typer, variabler	2.1,2.2,2.5.2 (s 31–37, 41–47, 69)
Typalias	2.5.1
Aritmetik	4.1-4.5, 4.11
Pekare och Referenser	2.3 (s 50–59)
Arrayer	3.5–3.5.2 (s 113–116)

Nästa föreläsning
Användardefinierade typer

Referenser till relaterade avsnitt i Lippman

Klasser	2.6, 7.1.4, 7.1.5, 13.1.3
std::string	3.2
std::vector	3.3
Uppräkningstyper	19.3
Scope och livstid	2.2.4, 6.1.1
I/O	1.2, 8.1–8.2
Operatoröverlagring	14.1