EDAF30 – Programmering i C++

2. Användardefinierade typer

Sven Gestegård Robertz
Datavetenskap, LTH

2016

Innehåll

Deklarationer
Scope

En deklaration inför ett *namn* i ett *scope* ("deklarationsområde"):

Local scope: Ett namn deklarerat i en funktion syns
- Från deklarationen
- Till slutet av blocket (block omges av `{ }`)
- Argument till funktioner är lokala namn

Class scope: Ett namn kallas en medlem (*member*) om det definieras i en klass. Det syns i hela klassen.

Namespace scope: Ett namn kallas *namespace member* om det definieras i ett namespace, utanför någon funktion eller, klass (eller *enum class*).
T ex `std::cout`.

Ett namn deklarerat utanför ovanstående kallas *globalt namn* och tillhör *the global namespace*.

Deklarationer
livstid

- Ett objekts livstid avgörs av dess *scope*:
  - måste initialiseras (konstrueras) innan det kan användas
  - förstörs (destrueras) vid slutet av dess *scope*.
  - *namespace*-objekt förstörs när programmet avslutas
  - Objekt som allokerats med **`new`** lever tills de förstörs med **`delete`**. (Skillnad mot Java)

Användardefinierade typer

- Inbyggda typer (t ex **`char`**, **`int`**, **`double`**, pekare, ...)
  - en direkt och effektiv representation av datorns funktionalitet
  - låg abstraktionsnivå
- Användardefinierade typer
  - abstraktion byggd med de inbyggda typerna
  - **`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)
- En *konkret typ* kan användas som en inbyggd typ

Strukturer

Första steget i att bygga en ny typ är att organisera dess element i en datastruktur (en *struct*).
Exempel: Punkt i 2D

```
struct Point{
    int x;
    int y;
};
```

En variabel av typen `Point` kan skapas med

```
Point p;
```

men medlemsvariablerna har odefinierade värden .

Vi måste *initialisera* variabeln

Strukturer
Initialisering

En funktion för att initiera en `Point`:

```
void init_point(Point& p, int x, int y)
  {
      p.x = x;
      p.y = y;
  }
```

En variabel av typen `Point`, kan då skapas med

```
Point p;
init_point(p, 10, 20);
```

- referensanrop: objektet `p` ändras

Strukturer
Användning

Nu kan vi använda vår typ `Point`:

```
#include <iostream>
Point read_point()
{
    cout << "Enter x coordinate:" << endl;
    int x;
    cin >> x;

    cout << "Enter y coordinate:" << endl;
    int y;
    cin >> y;

    Point p;
    init_point(p, x, y);
    return p;
}
```

- `>>` är *inmatningsoperatorn*
- standardbiblioteket `<iostream>`
- `std::cin` är *standard input*

Strukturer
Användning (2)

```
int square(int x)
{
  return x*x;
}

double distance(const Point& p1, const Point& p2)
{
  return sqrt( square(p2.x - p1.x) + square(p2.y - p1.y));
}

void example()
{
  Point p1 = read_point();
  Point p2 = read_point();
  cout << "The distance between points is " <<
          distance(p1,p2) << endl;
}
```


Klasser

- Ofta vill man kapsla både data och operationer tillsammans
- Få en typ att uppföra sig som en inbyggd typ
- Ofta: göra representationen oåtkomlig för användare

En klass har

- datamedlemmar (“attribut”)
- medlemsfunktioner (“metoder”)
- medlemmar kan vara
  - **public**
  - **private**
  - **protected**
  - som i Java

Klasser
Exempel

```
class Point{
public:
    Point(int xi=0, int yi=0) :x{xi}, y{yi} {}  // konstruktor
    double distance_to(const Point&);
    int get_x() {return x;}
    int get_y() {return y;}
private:
    int x;
    int y;
};
```

- *konstruktor*, som i Java
  - Skapar ett objekt och *initierar medlemmar*
  - satserna `Point p;` `init_point(p, 10, 20);` blir nu `Point p(10, 20);`
    - NB! p är ett *objekt*, inte en referens som i Java
  - Default-värden på parametrarna: `Point p;` blir `Point p(0,0);`
    → alltid initierade medlemsvariabler

Klasser
Exempel

```
class Point{
public:
    Point(int xi=0, int yi=0) :x{xi}, y{yi} {}
    double distance_to(const Point&);
    int get_x() {return x;}
    int get_y() {return y;}
private:
    int x;
    int y;
};

double Point::distance_to(const Point& p)
{
    return sqrt( square(p.x - x) + square(p.y - y));
}
```

- *medlemsfunktion*, (metod i Java)
  - deklareras i klassdefinitionen
  - definieras i eller utanför klassdefinitionen

Klasser
Exempel

```
Point read_point()
{
    cout << "Enter x coordinate:" << endl;
    int x;
    cin >> x;
    cout << "Enter y coordinate:" << endl;
    int y;
    cin >> y;
    return Point(x, y);
}


void example()
{
    Point p1 = read_point();
    Point p2 = read_point();
    cout << "The distance between points is " <<
            p1.distance_to(p2) << endl;
}
```

NB! return-by-value – returnerar (en kopia av) ett `Point`-objekt inte en `Point`-referens som i Java

Klassdefinitioner
Deklarationer och definitioner av medlemsfunktioner

Medlemsfunktioner (⇔ metoder i Java)

Definition av klass

```
class   Foo {
public:
    int fun(int, int);    // Deklaration av medlemsfunktion

    int times_two(int x) {return 2*x;} // ... inkl definition
};
```

Obs! Semikolon efter klass

Definition av medlemsfunktion (utanför klassen)

```
int Foo::fun(int x, int y) {
    // ...
}
```

Inget semikolon efter funktion

Filstruktur för klasser

Klassdefinitionen läggs i en headerfil (.h eller .hpp)

- För att inte riskera att definiera en klass mer än en gång används *include guards*:

```
#ifndef FOO_H
#define FOO_H
//...
class Foo {
//...
};
#endif
```

Medlemsfunktioner läggs i en källkodsfil (.cpp)

Uppräkningstyper
C-stil

`enum`: en mängd namgivna värden

```
enum ans {JA, NEJ, KANSKE, VET_EJ};
enum colour {BLUE=2, RED=3, GREEN=5, WHITE=7};

colour fgcol=BLUE;
colour bgcol=WHITE;
ans svar;

fgcol=RED;
bgcol=GREEN;
svar = NEJ;

fgcol = KANSKE; // error: cannot convert 'ans' to 'colour'
svar = 2;       // error: invalid conversion from 'int' to 'ans'

bool dumt = (fgcol == svar); // Tillåtet, ger kanske en varning

int x = fgcol;  // OK, x = 3
```


Uppräkningstyper
C++: `enum class`

Problem med `enum`

Namnen "läcker" ut i omgivande *scope*.

```
enum eyes {brown,  green, blue};
enum traffic_light {red, yellow, green};
```

```
error: redeclaration of 'green'
```

C++:`enum class`

```
enum class EyeColour {brown,  green, blue};
enum class TrafficLight {red, yellow, green};

EyeColour e;
TrafficLight t;

e = EyeColour::green;
t = TrafficLight::green;
```


A propos "namn-läckage"

I stället för

```
using namespace std;
```

är det ofta bättre vara specifik:

```
using std::cout;
using std::endl;
```

jfr Java:

```
import java.util.*;

import java.util.ArrayList;
```

Uppräkningstyper
Kommentarer

- **enum class**
  - En **enum class** implementerar alltid
    - initiering, tilldelning och jämförelseoperatorer (t ex == och <)
    - andra operatorer kan implementeras
  - Har ingen implicit konvertering till **int**
- **enum**
  - Värdena *är* heltal
- Ha ett värde som betyder "fel" eller "oinitierad".
  - det första värdet, om möjligt
  - initiera alltid variabler, annars blir det *odefinierat*
- Använd **enum class** om möjligt

Uppräkningstyper
Initiering

Deklarationer

```
enum alternatives {ERROR, ALT1, ALT2};
enum class alternatives2 {ERROR, ALT1, ALT2};
```

Variablerna får väldefinierade värden

```
alternatives  a{};
alternatives  b{ALT1};

alternatives2 p{};
alternatives2 q{alternatives2::ALT1};
```

Variablerna kan få vilka värden som helst

```
alternatives  x;
alternatives2 y;
```


Två typer ur standardbiblioteket
Alternativ till C-arrayer

Använd inte C-arrayer om du inte måste.
I stället för

- **char**[] – Strängar – använd `std::string`
- `T[]` – Sekvenser – använd `std::vector<T>`

Mer likt Java:

- mer funktionalitet – *"uppför sig som en inbyggd typ"*
- skyddsnät

Strängar: `std::string`

`std::string` har operationer för

- tilldelning
- kopiering
- konkatenering
- jämförelse
- in- och utmatning (<< >>)

och

- känner till sin storlek

Liknar `java.lang.String`

Exempel: `std::string`

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

string make_email(string fname,
                  string lname,
                  const string& domain)
{
    fname[0] = toupper(fname[0]);
    lname[0] = toupper(lname[0]);
    return fname + '.' + lname + '@' + domain;
}

void test_string()
{
    string sr = make_email("sven", "robertz", "cs.lth.se");

    cout << sr << endl;
}
```

```
Sven.Robertz@cs.lth.se
```


Sekvenser: `std::vector<T>`

En `std::vector<T>` är

- en ordnad samling objekt (av samma typ, `T`)
- varje element har ett index

som, till skillnad från en C-array

- känner till sin storlek
  - `vector<T>::operator[]` kollar inte gränser
  - `vector<T>::at(size_type)` kastar `out_of_range`
- kan växa (och krympa)
- kan tilldelas, jämföras, etc.

Liknar `java.util.ArrayList`

Är en *klassmall (class template)*

Exempel: std::vector<int> initiering

```
void print_vec(std::string s, std::vector<int> v)
{
    std::cout << s << " : " ;
    for(int e : v) {
        std::cout << e << " ";
    }
    std::cout << std::endl;
}
void test_vector_init()
{
    std::vector<int> x(7);
    print_vec("x", x);

    std::vector<int> y(7,5);
    print_vec("y", y);

    std::vector<int> z{1,2,3};
    print_vec("z", z);
}
```

```
x: 0 0 0 0 0 0 0
y: 5 5 5 5 5 5 5
z: 1 2 3
```


Exempel: std::vector<int> tilldelning

```
void test_vector_assign()
{
    std::vector<int> x {1,2,3,4,5};
    print_vec("x", x);
    std::vector<int> y {10,20,30,40,50};
    print_vec("y", y);
    std::vector<int> z;
    print_vec("z", z);
    z = {1,2,3,4,5,6,7,8,9};
    print_vec("z", z);
    z = x;
    print_vec("z", z);
}
```

```
x : 1 2 3 4 5
y : 10 20 30 40 50
z :
z : 1 2 3 4 5 6 7 8 9
z : 1 2 3 4 5
```


Exempel: std::vector<int> tilläggning och jämförelse

```
void test_vector_eq()
{
    std::vector<int> x {1,2,3};
    std::vector<int> y;
    y.push_back(1);
    y.push_back(2);
    y.push_back(3);

    if(x == y) {
        std::cout << "equal" << std::endl;
    } else {
        std::cout << "not equal" << std::endl;
    }
}
```

```
equal
```


Konsoll-I/O
Programexempel

In- och utmatning

```
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

int main( ) {
   string namn;

   cout << "Vad heter du? ";
   cin >> namn;
   cout << "Goddag, " << namn << "!" << endl;

   return 0;
}
```


Inmatning
från *standard in* – std::cin

Läs och returnera nästa tecken

```
std::cin.get()
```

Läs en rad tecken, returnera en std::string

```
std::string str;
std::getline(std::cin,str);
```

Läs ett antal tecken till C-strängen str

```
char str[STORLEK];              // OBS! str måste vara minst
std::cin.getline(str, STORLEK); // STORLEK tecken lång
```

Läs nästa tecken och släng det

```
std::cin.ignore()
```


Inmatning
std::getline() – fri funktion

Exmpel: läs till en std::string

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

int main()
{
    std::string str;
    std::getline(cin, str);
    cout << "read a string: " << str << endl;

}
```

Exekvering

```
abcdefghijkl
read a string: abcdefghijkl
```

std::string ändrar sin storlek så att strängen får plats

Inmatning
`cin.getline()` – medlemsfunktion i std::istream

Exmpel: läs till en char-array

```
#include <iostream>

int main()
{
    constexpr int n = 10;
    char str[n];              // allokera char-array med längd n

    std::cin.getline(str, n); // läs max n-1 tecken

    std::cout << "Read: " << str << std::endl;
}
```

Exekvering

```
abcdefghijkl
Read: abcdefghi
```

Skriver max $n-1$ tecken plus *null* till `str`

Filhantering

```
#include <fstream>
```

Läsa från fil

```
ifstream f1("infil.txt");

string ord;
f1 >> ord;
...

f1.close();   ~~f1.close();~~ Ska normalt inte anropas, görs av destruktorn i ifstream
```

Skriva till fil

```
ofstream f2("utfil.txt");
f2 << namn << " " << telnr << endl;
...

f1.close();   ~~f2.close();~~
```


Filhantering
Läsa/skriva tecken för tecken

Läsa oformaterat från fil

```
void print_bytes(std::string infile)
{
    std::ifstream in{infile};

    int c;
    while((c = in.get()) != EOF ) {
        std::cout <<  c << " [" << char(c) << "]\n";
    }
}
```


Operatoröverlagring

- Operatorer kan överlagras
  - som medlemsfunktioner
  - som fria funktioner

En användardefinierad typ kan användas som en inbyggd typ

Operatorer
Exempel

```
class Point{
public:
   Point(int xi=0, int yi=0) :x{xi}, y{yi} {}
   double distance_to(const Point&);
   int get_x() {return x;}
   int get_y() {return y;}
   bool operator==(const Point& p) {return (x == p.x) && (y == p.y);}
private:
   int x;
   int y;
};

void example_operators()
{
   Point p1{1,2};
   Point p2{1,2};
   Point p3{1,3};

   cout << "p1 == p2: " << (p1 == p2) << endl;
   cout << "p1 == p3: " << (p1 == p3) << endl;
}
```


Operatorer
Exempel

```
bool operator!=(Point& p1, Point& p2) {return !(p1 == p2);}
```

- Implementera **operator**!= i termer av **operator**==.
- Kan vara fri funktion, **operator**== är **public**

Namnrymder **namespace**

- Avgränsa synligheten för namn
  - tydligare vilka funktioner/klasser/konstanter som hör ihop
  - minskar risken för namnkrockar
  - Jfr package i Java
- Åtkomst av namn i namnrymder
  - fullständigt kvalificerat namn vid användning: `namnrymd::namn`
  - selektivt med **`using`** `namnrymd::namn`
  - import av alla namn med **`using namespace`** `namnrymd`
    - undvik generellt, eller använd avgränsat
  - skriv aldrig **`using`**-direktiv i header-filer
- Namnrymder kan *utökas*
  - Utom (med några undantag) `std` (⇒ undefined behaviour)

Namnrymder **namespace**
Exempel

deklarationer (.h)

```
namespace foo {
  void test();
}

namespace bar {
  void test();
}
```

definitioner (.cpp)

```
using std::cout;
using std::endl;

namespace foo {
  void test()
  {
    cout << "foo::test()\n";
  }
}

void bar::test()
{
  cout << "bar::test()\n";
}
```

```
int main()
{
    foo::test();
    bar::test();
    using namespace foo;
    test();
}
```

```
foo::test()
bar::test()
foo::test()
```


Namnrymder **namespace**

- Den namnlösa namnrymden
  - endast synlig i filen där den deklarerats
  - används för att dölja saker (jfr **`static`** i C)
  - Se upp med namkrock med globala namn

```
namespace foo {
  void test()
  {
    cout << "foo::test()\n";
  }
}
namespace {
  void test()
  {
    cout << "::test()\n";
  }
}
```

```
int main()
{
  test();
  foo::test();
  ::test();
}
```

```
::test()
foo::test()
::test()
```

- Alternativa namn för namnrymder:

```
namespace test=my_namespace_for_testing_stuff;
```


Läsanvisningar

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