EDAF30 – Programmering i C++

Funktioner

Sven Gestegård Robertz
Datavetenskap, LTH

2015

Innehåll

Funktionsdeklaration

- En *deklaration* av en funktion introducerar ett namn och ger
  - typen för funktionens returvärde
  - antal parametrar och deras typer
- *Declare before use*
  - *header-filer*
  - `#include "myfeature.h"`
- En funktion måste deklareras innan den kan användas
  - *header-filer* innehåller funktionsdeklarationer
  - `#include "filnamn"` överallt där funktionerna används
- En *definition* av en funktion innehåller implementationen
  - Får bara finnas på ett ställe

Funktionsdeklaration (syntax)

```
<returtyp> <funktionsnamn> ( <parameterlista> );
```


Funktionsdeklaration

Exempel

```
void test();
int add(int, int);
int pow(int x, int exponent);
```

- Kompilatorn ignorerar parameternamn
- Ge namn om det ökar läsbarheten

Exempel

```
char *strncat(char *dest, const char *src, size_t n);
string concat(const string& head, const string& tail);
```


Funktionsdefinition

- 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;
}
```

Överlagring

Olika varianter av samma funktion:

- Samma namn men olika parameterlistor
- Kan inte skilja på enbart returtyp
- Måste vara entydigt
- Alternativ: *default*-värden på parametrar

Exempel:

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

void user()
{
  print(42);            // anropar print(int)
  print(17.8);          // anropar print(double)
  print("some text");   // anropar print(std::string)
}
```


Överlagring
Fel

Funktioner får inte skilja sig endast i returtyp

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

Användning måste vara entydig

- Kompilatorn gör implicit typkonvertering
- Om två kandidater är "lika bra" ges ett fel

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

void user2()
{
  print(0,0);  // inte entydigt
}
```

Om även funktionen

```
void print(int,int);
```

finns så matchar denna bäst, och anropet är OK.

Parametrar med defaultvärden

Funktioner kan ha variabelt antal parametrar utan att överlagring används genom att använda *defaultvärden*.

Exempel: Kast med tärning

```
#include <cstdlib>
#include <ctime>

void kasta(int antal=1, int sidor=6)
{
   srand(time(0));
   for (int i=0; i<antal; i++) {
      cout << rand()%sidor + 1 << " ";
   }
   cout << endl;
}

kasta();     //ger 1 slag med 6-sidig tärning
kasta(5);    //ger 5 slag med 6-sidig tärning
kasta(1,5); //ger 1 slag med 5-sidig tärning
```


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, y;
...
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)
{
   auto tmp=a;
   a = b;
   b = tmp;
}
```

Nu används *referenser* till de aktuella parametrarna istället för själva *värdena*. Därför byter verkligen talen plats denna gång.

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'
```


Värdeanrop och referensanrop
Pekaranrop

I vissa fall används *pekare* istället för *referenser*:

Exempel: Byta plats på två heltalsvärden

```
void swap2(int* a, int* b)
{
  if(a != nullptr && b != nullptr) {
    int tmp=*a;
    *a = *b;
    *b = tmp;
  }
}  ...  och användning:      int x, y;
                             ...
                             swap2(&x, &y);
```

Notera:

- en pekare kan vara `nullptr` eller oinitierad
- operatorn `&`: `&x` betyder "adressen till objektet `x`".
- operatorn `*`: `*p` betyder "inneållet i adressen `p`".

Mer om pekare senare i kursen.

Värdeanrop och referensanrop
Arrayer som parametrar

Det kan se ut som om arrayer referensanropas.

Exempel: Heltalsvektorns element kvadreras

```
void kvadrera(int vec[], int n)
{
  for (int i=0; i != n; ++i) {
     vec[i] = vec[i] * vec[i];
  }
}
... anrop:
  int x[] {1,2,3};
  kvadrera(x, 3);
```

- Samma prototyp som `void kvadrera(int* vec, int n);`
- En arrayvariabel som parameter *omvandlas implicit till en pekare till första elementet*. (*arrays decay into pointers*)
- Funktionen *värdeanropas* med en *pekare* som parameter.

Värdeanrop och referensanrop
Utparametrar

Referensanrop kan användas för att "returnera" värden via s.k. *utparametrar*.

- Mycket vanligt i C-program (med pekare)
  - För att returnera flera värden
  - jfr.`<cstdlib>`
- Behövs sällan i C++
  - Använd bara om parametern *har identitet* och *ska ändras*
  - *Return by value* är mycket tydligare och blir ofta effektivt
  - Risk att göra fel (t ex minnesallokering)
  - `std::pair`, `std::tuple`
  - Generellt: undvik prematur optimering

Deklarera referensparametrar som inte ändras `const`.

Värdeanrop och referensanrop
Utparametrar: exempel

Referensanrop kan användas för att "returnera" värden via s.k. *utparametrar*.

kvadrera två tal

```
void kvadrera(int& x, int& y)
{
   x = x*x;
   y = y*y;
}
```

Alternativ: returnera **`struct`**, `std::pair` eller `std::tuple`

`std::getline`

```
istream& getline (istream&  is, string& str);

istream& istream::getline (char* s, streamsize n );
```


Rekursiva funktioner

- En funktion som anropar sig själv (direkt eller indirekt) sägs vara *rekursiv*.
- Två fall:
  - basfall
  - rekursivt anrop

Exempel: Fakultet, $n!$

```
unsigned int fac(unsigned int n)
{
   if (n==0) return 1;
   return n * fac(n-1);
}
```

- Användning
  - Länkade strukturer
  - Söndra och härska (*divide and conquer*)

Rekursiva funktioner
Exempel: Binomialkoefficientberäkning

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad n,k \in \mathbb{N}$$
$$\binom{n}{0} = \binom{n}{n} = 1$$

```
int binom(int n, int k)
{
   if ( (k==0) || (n==k) )
      return 1;
   else
      return binom(n-1, k-1) + binom(n-1, k);
}
```

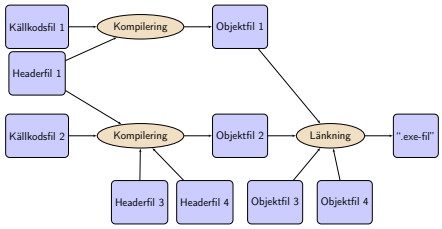
http://sv.wikipedia.org/wiki/Binomialkoefficient
http://sv.wikipedia.org/wiki/Pascals_triangel

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`**
- Separatkompilering
  - minimera kompileringstid i stora system
  - kompilera bara om den kod som ändrats
  - bibliotek

Uppdelning av program i flera filer

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



Uppdelning av program i flera filer
Exempel: Headerfil

Minimalt exempel: Uppdelning av medelv:

- medelv.h
- medelv.cpp
- main.cpp

medelv.h: deklarationer

```
double medelv(double, double);
double medelv(int, int);
double medelv(int*, int);
```


Uppdelning av program i flera filer
Exempel: Källfil

medelv.cpp: definitioner

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

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

double medelv(int x[], int n)
{
   double s=0;
   for (int i=0; i<n; i++) {
      s += x[i];
   }
   return s/n;
}
```


Uppdelning av program i flera filer
Exempel: Huvudprogram

main.cpp: användning

```
#include <iostream>
#include "medelv.h"     // gör deklarationer synliga

using namespace std;
int c[] = {3, 4, 3, 3, 4, 5, 4, 3, 5};

int main()
{
   double a=2.3, b=3.9;
   int m=3, n=4;
   cout << medelv(a, b) << endl;
   cout << medelv(m, n) << endl;
   cout << medelv(c, 9) << endl;
}
```


Sekvenser i C++

- En *sekvens* är en följd av element
- `std::vector`: variabel storlek, $O(1)$ indexering
- `std::array`: fast storlek, $O(1)$ indexering, lika effektiv som C-array, men säkrare
- `std::list`: variabel storlek, $O(1)$ insättning och borttagning, långsam ($O(n)$) indexering
- `std::deque`: variabel storlek, (effektivare än vector), $O(1)$ insättning och borttagning i början och slutet, $O(1)$ indexering.

Med *indexering* avses *random access*

Sekvenser: exempel
`std::array`

Exempel

```
array<char,12> a1 {"Hello Again!"};
```

```
error: initializer-string for array of chars is too long
```

Exempel

```
array<char,13> a1 {"Hello Again!"};

for (char ch : a1) {
   cout << "[" << c << "]";
}
cout << endl;
```

```
[H][e][l][l][o][ ][A][g][a][i][n][!][]
```

NB! utan avslutande \0 blir längden 12:

```
std::array<char,12> a2{'H', 'e', 'l', 'l', 'o', ' ',
                       'A', 'g', 'a', 'i', 'n', '!'};
```

Par

`std::pair`

Enkelt sätt att definiera par av olika datatyper

```
#include <utility>

pair<string, int> p1{"Nils", 42};

cout << p1.first << " " << p1.second << endl;

pair<string, int> p2 = p1;
p2.first = "Nisse";

cout << "----" << endl;
cout << p1.first << " " << p1.second << endl;
cout << p2.first << " " << p2.second << endl;
```

```
Nils 42
----
Nils 42
Nisse 42
```

Finns även `std::tuple` för tupler med > 2 element.

Heltalstyper

- heltal med tecken

Typ	Storlek	Talområde (minst)
signed char	8 bitar	$[-127, 127]^*$
short	minst 16 bitar	$[-2^{15}+1,\ 2^{15}-1]$
int	minst 16 bitar, oftast 32	$[-2^{15}+1,\ 2^{15}-1]$
long	minst 32 bitar	$[-2^{31}+1,\ 2^{31}-1]$
long long	minst 64 bitar	$[-2^{63}+1,\ 2^{63}-1]$

*typiskt $[-128, 127]$, etc.

- **unsigned**(icke-negativa) heltal
  - samma storlek som motsvarande **signed** typ
  - **unsigned char**: $[0, 255]$, **unsigned short**: $[0,\ 2^{16}-1]$. etc.
- specialfall
  - **char** (får *representeras* som **signed char** *eller* **unsigned char**)
  - Använd **char** endast för att lagra tecken
  - Använd **signed char** eller **unsigned char** för talvärden
- Storlekar enligt standarden:
  char $\leq$ **short** $\leq$ **int** $\leq$ **long** $\leq$ **long long**

Heltalstyper

Test med `sizeof`

```
#include <iostream>
using namespace std;
int main () {
   cout << "sizeof(char)= \t" << sizeof(char)<<endl;
   cout << "sizeof(short)= \t" << sizeof(short) <<endl;
   cout << "sizeof(int) = \t" << sizeof(int) <<endl;
   cout << "sizeof(long)= \t" << sizeof(long)<<endl;
   cin >> ws; // Bryt med Ctrl-C ...
}
```

```
sizeof(char)=   1
sizeof(short)=  2
sizeof(int) =   4
sizeof(long)=   8
```


Heltalstyper – Test av talområdet via casting

eller: *se upp med typecasts* från **signed** till **unsigned**-typer

```
int main () {
 cout << "(char) -1 = " << (int)(char) -1 << endl;
 cout << "(unsigned char) -1 = " << (int)(unsigned char) -1 << endl;
 cout << "(short int) -1 = " << (short int) -1 << endl;
 cout << "(unsigned short int) -1 = "<<(unsigned short int)-1<<endl;
 cout << "(int) -1 = " << (int) -1 << endl;
 cout << "(unsigned int) -1 = " << (unsigned int) -1 << endl;
 cout << "(long) -1 = " << (long) -1 << endl;
 cout << "(unsigned long) -1 = " << (unsigned long) -1 << endl;
 cin >> ws;
}
```

```
(char) -1 = -1
(unsigned char) -1 = 255
(short int) -1 = -1
(unsigned short int) -1 = 65535
(int) -1 = -1
(unsigned int) -1 = 4294967295
(long) -1 = -1
(unsigned long) -1 = 18446744073709551615
```


Heltalstyper

Storlekarna specificeras i `<climits>`

CHAR_BIT	Number of bits in a **char** object (byte) (>=8)
SCHAR_MIN	Minimum value for an object of type **signed char**
SCHAR_MAX	Maximum value for an object of type **signed char**
UCHAR_MAX	Maximum value for an object of type **unsigned char**
CHAR_MIN	Minimum value for an object of type **char** (either SCHAR_MIN or 0)
CHAR_MAX	Maximum value for an object of type **char** (either SCHAR_MAX or UCHAR_MAX)
SHRT_MIN	Minimum value for an object of type **short int**
SHRT_MAX	Maximum value for an object of type **short int**
USHRT_MAX	Maximum value for an object of type **unsigned short int**
INT_MIN	Minimum value for an object of type **int**
INT_MAX	Maximum value for an object of type **int**
UINT_MAX	Maximum value for an object of type **unsigned int**
LONG_MIN	Minimum value for an object of type **long int**
LONG_MAX	Maximum value for an object of type **long int**
ULONG_MAX	Maximum value for an object of type **unsigned long int**
LLONG_MIN	Minimum value for an object of type **long long int**
LLONG_MAX	Maximum value for an object of type **long long int**
ULLONG_MAX	Maximum value for an object of type **unsigned long long**

Heltalstyper

Storlekarna specificeras i `<climits>`

```
#include <iostream>
#include <climits>
int main()
{
    std::cout << CHAR_MIN << ", " << CHAR_MAX << ", ";
    std::cout << UCHAR_MAX << std::endl;
    std::cout << SHRT_MIN << ", " << SHRT_MAX << ", ";
    std::cout << USHRT_MAX << std::endl;
    std::cout << INT_MIN << ", " << INT_MAX << ", ";
    std::cout << UINT_MAX << std::endl;
    std::cout << LONG_MIN << ", " << LONG_MAX << ", ";
    std::cout << ULONG_MAX << std::endl;
    std::cout << LLONG_MIN << ", " << LLONG_MAX << ", ";
    std::cout << ULLONG_MAX << std::endl;
}
```

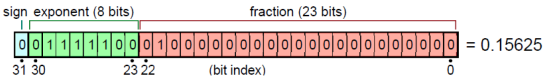
```
-128, 127, 255
-32768, 32767, 65535
-2147483648, 2147483647, 4294967295
-9223372036854775808, 9223372036854775807, 18446744073709551615
-9223372036854775808, 9223372036854775807, 18446744073709551615
```

Flyttalstyper

`float` (oftast 32 bitar) **`double`** (oftast 64 bitar) **`long double`**

IEEE 754 32-bitars flyttal: $1 + 8 + 23$ bitar
(tecken + exponent + mantissa)

värde $= -1^{sign} \cdot (1.b_{22}b_{21}\ldots b_0)_2 \cdot 2^{(b_{30}b_{20}\ldots b_{23})_2 - 127}$
$= -1^{sign} \cdot \left(1 + \sum_{i=1}^{23} b_{23-i} \cdot 2^{-i}\right) \cdot 2^{exponent-127}$



exponent $=$ `0x7c` $= 124$

värde $= +1.01_2 \cdot 2^{124-127} = 1.25 \cdot 2^{-3} = 1.25 \cdot 0.125$

Flyttalstyper

Test med `sizeof`

```
#include <iostream>
using namespace std;
int main () {
   cout << "sizeof(float)="
        << sizeof(float)<<endl;
   cout << "sizeof(double)="
        << sizeof(double) << endl;
   cout << "sizeof(long double)="
        << sizeof(long double) << endl;
   cin >> ws; // Bryt med Ctrl-C ...
}
```

```
sizeof(float)=4
sizeof(double)=8
sizeof(long double)=12
```


Sammanfattning

Vi har talat om

- Funktioner
  - Deklaration och definition, överlagring
  - Defaultvärden för parametrar
  - Värde- och referensanrop
  - Rekursion
- Separatkompilering
- Typer för talvärden

Nästa föreläsning:

Vi kommer att gå igenom

- Typer
  - pekare
  - arrayer (“fält”)
  - uppräkningstyper (**`enum`**)
- Minnesallokering
- Typomvandling (*type casting*)