

# Inlupp 2: pekare och funktioner

Imperativ Programmering DT501G, HT 2025  
2 december 2025

De här uppgifterna handlar framför allt om att hantera pekare, strängar och fält, samt funktioner. Föreläsning 3–4 tar upp det som krävs för uppgift 2.1–2.8, och föreläsning 5 tar upp filer, som krävs för uppgift 2.9–2.11.

I exemplen som visar hur dina program ska bete sig är input och output färglagt enligt nedan:

```
input, som du skriver in
output, som programmet skriver ut
```

## Lärandemål

- Funktionsanrop i C.
- Felsökning (debugger).
- Inargument till ett program via `argv` och `argc`.
- Förstå skillnaden på värdeanrop och pekaranrop, och kunna använda dem på rätt sätt.
- Kunna hantera `const` för pekare och andra variabler.
- Definiera egna typer med `typedef`.
- Kunna använda sammansatta datatyper, `struct`.
- Kunna hantera och traversera strängar och fält.
- Kunna definiera och använda testfall.

## Frågor

Frågorna som behöver besvaras är markerade i marginalen. Precis som tidigare ska du skriva en textfil `svar.txt` där du svarar på frågorna (fråga 1–4), och lägga till den i ditt git-repo, tillsammans med källkoden.

## Om testfall och felhantering

För alla uppgifter här gäller att du måste kunna hantera, till viss del, felaktig inmatning från användaren. Framför allt behöver du se till att du får rätt antal argument vid inmatning, och att programmet inte kraschar om det kommer något oväntat. Se till att hantera det på ett vettigt sätt (till exempel genom att skriva ut ett felmeddelande, och avsluta programmet med en felkod). Däremot måste du inte hantera om användaren matar in bokstäver i stället för siffror eller vice versa.

Glöm inte det här!

För varje uppgift, utom 2.8–2.11 specificera minst två testfall som du har använt för att kontrollera att programmet gör som det ska. De båda testfallen ska testa olika saker (inte bara två varianter av giltigt input, t ex). Ange dina testfall, och förväntat resultat, som kommentarer i din källkodsfil.

Exempel för uppgift 2.1:

```

//...
int main( int argc, char *argv[] )
{
    // ...
}

/* TESTFALL
Test med heltal och flyttal
input: task_1 3 2.5
resultat OK: 5.5

Test med negativa tal
input: task_1 -8 -5
resultat OK: -13

Test med ogiltig input
input: task_1 0
resultat: Programmet avslutas med ett felmeddelande som säger att
mer input behövs.
*/

```

## Dokumentation

Glöm inte det här!

Alla funktioner ska dokumenteras enligt de givna instruktionerna (se “Generella instruktioner”). Dokumentera också större kodblock så att det framgår vad det är tänkt att de ska göra.

## Uppgifter

### 2.1 En funktion

Skriv ett program med en funktion `double sum( double a, double b )` som returnerar summan av talen a och b. Ditt program ska läsa in två tal från kommandoraden (via `argv`) och skriva ut summan. Du kan använda funktionen `strtod`<sup>1</sup> från `stdlib.h` för att konvertera från en sträng från `argv` till ett flyttal. Så länge vi bara får in siffror som inmatning går det bra att skicka `NULL` som andra argument till `strtod`, det vill säga `double value = strtod( argv[i], NULL );` Det andra argumentet används för felkontroll, och vi kommer till det i inlupp 3.

```

./task_1 3 -2.5
0.500000

```

### 2.2 En funktion med pekare

Skriv ett program med en funktion `sum` som, i stället för att returnera summan, skriver det till en av parametrarna. För att det här ska fungera behöver du alltså ha (minst) en parameter som är en pekare. Från användarens perspektiv ska det här programmet vara identiskt med det i den förra uppgiften.

Fråga 1

Använder du `const` för någon eller några av dina pekare? Motivera.

---

<sup>1</sup>Se <http://www.fortran-2000.com/ArnaudRecipes/Cstd/2.13.html#strtod>

### 2.3 Fält

Skriv ett program som läser in 3 numeriska värden från kommandoraden, lagrar dem internt i ett `double`-fält som rymmer 3 värden, och skriv sedan ut värdena baklänges, med blanktecken emellan.

Exempel:

```
./task_3 1 3 5.6  
5.600000 3.000000 1.000000
```

### 2.4 Vektorlängd

Skriv ett program som läser in 3 numeriska värden från kommandoraden, och lagrar dem internt i ett `double`-fält som rymmer 3 värden. Programmet ska innehålla en separat funktion som beräknar längden på en 3-vektor, det vill säga  $l = \sqrt{x^2 + y^2 + z^2}$ .

$$(1, 3, 5.6) \quad l = \|(1, 3, 5.6)\| = 6.431$$

Låt ditt program skriva ut längden av den inmatade vektorn. Utskriften ska ske i `main` och din funktion ska bara returnera värdet.

```
./task_4 1 3 5.6  
6.431174
```

Du kan antingen låta din funktion ta tre flyttal som parametrar, eller ett fält. (Fundera på om det finns någon praktisk skillnad mellan de två alternativen. Är det ena mer effektivt än det andra, när det gäller minnesanvändning? Tips: kolla inne i din funktion hur många bytes varje `double` använder med hjälp av `sizeof(double)`, jämfört med en pekare `sizeof(double*)`.)

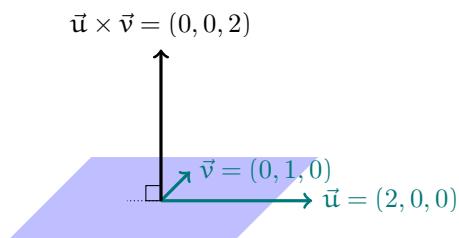
### 2.5 Vektorprodukt

Skriv nu ett program med en funktion som beräknar vektorprodukten (kryssprodukten) av två vektorer, och skriver ut resultatet. Vektorprodukten för två vektorer  $(x, y, z) = (x_1, y_1, z_1) \times (x_2, y_2, z_2)$  beräknas så här:

$$x = y_1 z_2 - z_1 y_2$$

$$y = z_1 x_2 - x_1 z_2$$

$$z = x_1 y_2 - y_1 x_2$$



Din funktion ska ta tre fält som argument: ett för varje vektor.

I det här fallet är det svårt att returnera resultatet från funktionen som beräknar produkten, eftersom det bara går att returnera ett värde i C (inga tupler, och inga fält). Alltså måste du använda ett ut-argument, precis som du gjorde i uppgift 2.2.

I din funktion, var noga med att ange `const` på ett korrekt sätt för dina parametrar. Pekarargument där inget ändras ska generellt sett deklaras med `const`.

Exemplet nedan visar hur det ska se ut. Programmet läser alltså in två vektorer  $\vec{u} = (1, 2, 3)$  och  $\vec{v} = (4, 5, 6)$  och skriver ut  $\vec{u} \times \vec{v}$ .

```
./task_5 2 0 0 0 1 0  
0.000000 0.000000 2.000000
```

I den här deluppgiften, och endast här, ska du inte kolla i ditt program att du får in tillräckligt antal argument i `argv`. Kör nu ditt program med bara tre inputvärdet. Då kommer det antagligen att krascha. Stega igenom programmet med en avlusare (debugger). Se avsnitt 4.3 i "Snabbkurs i Linux, GCC och Git". För att kunna köra avlusaren igenom programmet är det viktigt att kompileringen görs med flaggan `-g`, enligt avsnitt 4.2.1–4.2.2 i "Snabbkurs i Linux, GCC och Git".

#### Fråga 2

På vilken rad kraschar programmet? Och varför?

#### 2.6 Rimstuga

Skriv ett program som tar två strängar som argument och avgör om de rimmar eller ej. För den här uppgiften använder vi en väldigt lös definition på att rimma: det räcker om de tre sista bokstäverna är samma. (Tips: använd gärna `strlen` från biblioteket `string.h` för att kolla längden på strängar.)

Även i den här uppgiften ska själva funktionaliteten, att avgöra om två strängar rimmar eller ej, ligga i en egen funktion, och inte i `main`.

Ditt program ska dels skriva ut resultatet på skärmen ("ja" eller "nej"), och dels returnera ett värde. Till skillnad från alla andra funktioner så returnerar `main` värdet 0 även om du inte har med en `return 0;`; men här ska ditt program returnera

- 0, om inargumenten rimmar, eller
- 1, om de inte rimmar, eller
- en annan felkod om något gick fel (tex om användaren matade in för få ord, eller om de inte innehåller minst tre bokstäver).

Exempel:

```
./task_6 imperativ programmering  
nej  
  
./task_6 programmering schmogrammering  
ja
```

#### 2.7 Kopiera strängar

Funktionen `strcpy`, från standardbiblioteket `string.h`, kopierar en sträng (inklusive sluttecknet) till en annan. Den här uppgiften går ut på att skriva en egen `strcpy`-funktion, utan att använda standardfunktionen.<sup>2</sup>

Skriv en funktion som kopierar en sträng enligt följande mall.

```
/*  
 * Kopiera en sträng till en annan. Inkluderar sluttecknet, och  
 * avslutar kopieringen där.
```

<sup>2</sup><http://www.fortran-2000.com/ArnaudRecipes/Cstd/2.14.html#strcpy>

```

/*
 * Parametrar:
 * destination Pekar på strängen där resultatet ska hamna.
 * source      Pekar på input-strängen.
 *
 * Returnerar:
 * Antalet tecken som har kopierats (inklusive
 * avslutningstecknet).
 */
int string_copy( char *destination, const char *source )
{
    /*...*/
}

```

(Kanske kan det tyckas bakvänt att ha destinationssträngen till vänster, men det följer ju samma form som vanlig tilldelning, och även `strcpy`. Notera att returnvärdet är en `int` för den här funktionen, och inte en `char*` som det är i `strcpy`.)

Använd din funktion i ett program som läser in två strängar med `scanf`, kopierar den ena till den andra, och skriver ut dem igen.

```

./task_7
hejsan
svejsan
hejsan hejsan

```

Kopiering av strängar är en notorisk felkälla (och något som också har gett upphov till flera så kallade ”exploits”<sup>3</sup>). Vad kan hända om om fältet som `source` pekar på är större än fältet som `destination` pekar på? Testa att försöka provocera fram ett fel genom att köra följande i din `main`.

```

1  char X[2] = "a";           // En kort sträng som innehåller ['a', '\0']
2  char Y[1024];            // En lång sträng
3  for (int i = 0; i < 1023; ++i)
4      Y[i] = 'A'+i%26;     // Lägg in "ABCDE..." o.s.v. i strängen Y
5  Y[1023] = 0;              // Se till att Y avslutas korrekt med '\0'
6  printf( "X: %s\nY: %s\n\n", X, Y ); // Skriv ut strängarna
7  string_copy( X, Y );        // Kopiera
8  printf( "X: %s\nY: %s\n\n", X, Y ); // Skriv ut efter kopiering

```

Det är inte säkert att koden ovan gör samma sak varje gång den körs, men antagligen kommer även strängen `Y` att ha ändrats, efter att den kopieras till `X` på rad 5.

**Fråga 3** Kan du förklara vad som händer? Varför ändras även `Y` efter att `string_copy` har körts, fastän det är `X` som är destination? Varför är `X` nu längre än 1 tecken lång, fastän `X[2]` bara har plats för en bokstav (plus '`\0`')?

(Du behöver inte implementera någon avancerad felkontroll för att hantera de här fallen, men fundera bara på vad som kan hänta.)

## 2.8 En datastruktur för bilder

De nästföljande uppgifterna handlar om bildmanipulation.

Börja med att skapa en `struct` som innehåller den information som behövs om en bild.

---

<sup>3</sup>Se tex den här länken och den här om du är intresserad.

Vi kommer att använda gråskalebilder, så det som är användbart är dels ett fält med pixelvärdet (hur ljus varje pixel är), dels data om hur hög och bred bilden är, och dels data som anger maxvärdet på en pixel (alltså, vilket värde motsvarar helvitt).

Med andra ord, gör en `struct` som innehåller tre heltalsvariabler (en som kan lagra bildens bredd, en för höjden och en för maxvärdet på pixlarna), samt ett fält med heltal där pixelvärdena ska ligga. Se till att definiera fältet så att det har plats för tillräckligt många pixlar. Du kan utgå från att du inte kommer att hantera några bilder som är större än  $500 \times 500$  pixlar. Man kan tänka sig antingen ett vanligt endimensionellt fält för att lagra pixlarna (tex `int[]`), eller ett tvådimensionellt fält (tex `int[][]`). (Vilket föredrar du? Varför?)

Gör också en `typedef` så att du enkelt kan skapa variabler av den här `struct`-typen i ditt program. Alltså:

```
typedef struct {
    // fyll i här
} image;
```

Den här datatypen `image` kommer du sedan att använda i alla de kommande uppgifterna.

Innan du faktiskt ska läsa in en bild och göra något (i uppgift 2.9) så ska du kontrollera vad som händer om du skickar din `struct` som en kopia eller via en pekare.

Skapa två funktioner:

```
void image_thrash_by_value( image I, unsigned int c )
{
    printf( "%u\n", c );
    image_thrash_by_value( I, c+1 );
}

void image_thrash_by_pointer( const image * I, unsigned int c )
{
    printf( "%u\n", c );
    image_thrash_by_pointer( I, c+1 );
}
```

De här funktionerna gör alltså ingenting vettigt! De bara anropar sig själva rekursivt och skriver ut en räknare, för att vi ska se poängen med att skicka pekare till stora variabler.

Modifera funktionerna ovan så att de förutom `c` också skriver ut bredd och höjd på den `image` som kommer in. Skapa sedan en variabel av din typ `image`, fyll i ett värde för bredd och höjd (tex 500 och 500). Testa sedan vad som händer när du kör `image_thrash_by_value( I, 0 )` respektive `image_thrash_by_pointer( &I, 0 )`.

Båda funktionerna kommer att skriva ut samma värden. Däremot kommer ditt program att krascha så småningom, i båda fallen.

Vilken version kör längst? Varför? (Tips: Det har att göra med storleken på anropsstacken, som togs upp i föreläsning 3.)

För den här uppgiften behöver du inte ange några testfall i din inlämning.

Fråga 4



(a) Original

(b) Trösklad

(c) Suddig

Figur 1: Dansbandet Garvis i olika versioner.

## 2.9 Bildbehandling

Nu ska du skriva ett program som läser in en bildfil, till exempel den i fig. 1a. Du kommer att läsa och skriva bilder på formatet PGM, som är ett väldigt enkelt bildformat.<sup>4</sup>

Att läsa och skriva filer tas upp på föreläsning nr 5 (kapitel 10).

PGM-filer är textfiler som börjar med strängen "P2", sedan två heltal som anger bredd och höjd på bilden, efter det ett tal som anger maxvärdet för en pixel (om maxvärdet är 255 betyder det att svarta pixlar har värdet 0 och vita har värdet 255). Efter dessa värden (som kallas header) anges bildens pixlar, med en siffra per pixel, som anger hur ljus den är. En PGM-fil kan också innehålla kommentarer, som börjar med #, men det hoppar vi över i den här uppgiften.

Öppna filen `garvis.pgm` (eller någon av de andra PGM-bilderna som följer med uppgiften) i en texteditor så ser du hur filen ser ut. Du kan också dubbelklicka på `garvis.pgm` i filhanteraren så ser du hur den ser ut som bild. Det finns också två små bilder som är med som exempel, och som är bra att testa din kod på: `gradient.pgm` och `border.pgm`. De är bara  $7 \times 5$  pixlar stora. Kolla på dem också i en texteditor och med bildvisaren från filhanteraren.

Första uppgiften är att läsa in en PGM-fil. Ditt program ska ta ett filnamn som in-argument, och läsa in den filen (förutsatt att den finns).

Använd din datastruktur `image` från den föregående uppgiften. Skapa en variabel med typen `FILE*` för att läsa in filen. Använd `fopen` och `fscanf` för att läsa in den och fylla i en `image`-variabel. Se till att sluta läsa (`fscanf`) när filen tar slut. Skriv ut, med `printf`, värdet på den första och sista pixeln (på samma rad med mellanslag emellan), för att kontrollera att du har läst in rätt värden.

Glöm inte att stänga filen med `fclose` när du är färdig.

Exempel:

```
./task_9 garvis.pgm
160 9
```

För den här och de kommande uppgifterna som handlar om bildbehandling så behöver du inte ange vilka testfall du har kört. Testa med exempelbilderna som ges, och kolla att du får samma pixelvärdet som anges i fig. 2 och 3.

## 2.10 Tröskling

Nu ska du (äntligen) skriva ett litet bildbehandlingsprogram faktiskt gör något. Du ska skriva ett program som trösklar en bild, så att den ser ut som i fig. 1b.

<sup>4</sup>Se <http://netpbm.sourceforge.net/doc/pgm.html>

0	25	50	100	150	200	250	0	0	0	0	255	255	255
0	25	50	100	150	200	250	0	0	0	0	255	255	255
0	25	50	100	150	200	250	0	0	0	0	255	255	255
0	25	50	100	150	200	250	0	0	0	0	255	255	255
0	25	50	100	150	200	250	0	0	0	0	255	255	255

(a) Original

(b) Trösklad (2.10)

Figur 2: Testbilden `gradient.pgm`

Operationen går helt enkelt till så att alla pixlar som har ett värde mindre än ett visst tröskelvärde sätts till 0, och alla andra pixlar sätts till maxvärdet (255). Figur 2b visar ett exempel på testbilden `gradient.pgm` före och efter tröskling, med pixelvärdet.

Nu behöver ditt program skapa en till `FILE*` för den trösklade bilden, och öppna den med `fopen` för skrivning. Gå sedan igenom din bild-datastruktur och skriv ut en korrekt PGM-fil som är trösklad på värdet 120. (Alltså: pixlar som har värdet 120 och över sätts till 255, och de som har 119 och lägre sätts till 0.) Läs in filnamnet från kommandoraden som i föregående uppgift.

Skriv den nya bilden till en fil som ska heta `threshold.pgm`.

Din tröskling ska utföras i en egen funktion, och inte i `main`. Du ska alltså skriva en funktion som tar in en bild enligt din datastruktur, och skickar ut en trösklad version av samma bild. Den här funktionen ska inte skriva till fil. Att skriva resultatet till `threshold.pgm` gör du i stället i `main` (eller, ännu snyggare, i en separat funktion som bara har som syfte att skriva bildfiler).

Du ska inte använda en global variabel för din bild. Du måste skapa den i `main` och skicka den som argument till dina övriga funktioner.

(I din funktion, väljer du att skicka pekare till `image` som in- och ut-argument, eller väljer du att returnera en `image`? Vilket tycker du är att föredra, och varför?)

För denna och nästa uppgift är det klart lättast att använda de små testbilderna `gradient.pgm` och `border.pgm` och titta på sifervärdena som du skriver ut i din output-bild. De större bilderna med lite roligare motiv kan du testa med när det verkar som att ditt program funkar.

## Veckans mästarprov

### 2.11 Sudda

Slutligen ska du skriva ett program som gör bilden suddig, som i fig. 1c. Det här är lite knepigare, eftersom du måste jämföra pixlar som ligger bredvid varandra. Ut-värdet på varje pixel  $p$  du skriver ska nu vara medelvärdet av pixlarna som ligger till vänster, höger, ovanför och nedanför  $p$ , samt  $p$  själv. Då behöver du leta på rätt ställe i fältet där du har lagrat in-pixelvärdena. Om du har ett endimensionellt fält kommer då pixeln på raden under  $p$  ligga på plats  $p + w$ , där  $w$  är bildens bredd.

Viktigt här är att se till att du inte försöker läsa på en plats som ligger utanför fältet, vilket är lätt hänt, framför allt på första och sista raden i bilden! (Tips: debug-verktyget i Geany är användbart här, för att bena ut vad som händer om det inte blir som du tänker dig!)

Du behöver hantera första och sista raden lite annorlunda än de andra. Det finns olika strategier för det här, men i den här uppgiften ska du hoppa över de värden som inte finns. Med andra ord ska du beräkna medelvärdet av  $(p + \text{vänster}$

0	0	0	0	0	0	50
150	255	255	255	255	255	50
150	255	255	255	255	255	50
150	255	255	255	255	255	50
150	100	100	100	100	100	100

50	63	63	63	76	33
138	183	204	204	204	163
176	234	255	255	255	214
176	203	224	224	224	183
133	151	138	138	138	83

(a) Original

(b) Suddig (2.11). De pixlar som används för att beräkna pixeln i hörnet är markerade i grönt. De pixlar som används för att beräkna pixeln i mitten är markerade i blått.

Figur 3: Testbilden `border.pgm`

+ höger + upp + ned) för pixlar som ligger mitt i bilden, men (p + vänster + höger + upp) för den nedersta raden. Även den vänstra och högra pixelkolumnen behöver hanteras separat, jämfört med pixlarna mitt i bilden. Gör på samma sätt här som med den översta och understa raden i bilden. Med andra ord ska du inte bara läsa vidare och ta en pixel från föregående eller nedanstående rad.

Figur 3 visar testfilen `border.pgm` före och efter suddning. Du kan ha de här värdena som referens när du gör ditt program. Mittenpixeln i den suddade bilden har alltså värdet  $(255 + 255 + 255 + 255 + 255)/5 = 255$ , och övre vänstra hörnet är  $(0 + 0 + 150)/3 = 50$ .

Suddningsfunktionaliteten ska ligga i en egen funktion, och inte i `main`, precis som i föregående uppgift. Du får heller inte använda en global variabel för din bild, utan måste skicka den som argument till dina övriga funktioner.

Skriv den suddiga bilden till `blur.pgm`. Den ska ha samma antal pixlar som input-bilden.