Intro till C++

för dataIntro

Den 27:a september 2021

Innehåll

Kapitel 0

Programmeringslabbar i dataIntro!

Detta labhäfte innehåller den introduktion till programmering som ingår i *datateknisk introduktionskurs*. Labhäftet består av de inledande kapitlen till det större labhäfte som kommer att användas av DIS- och DMP-studenterna i en senare kurs.

0.1 Qt Creator/Visual Studio

När man programmerar behöver man egentligen bara ha en texteditor och en kompilator, men ofta brukar man använda en *integrerad programutvecklingsmiljö (eng. IDE,* integrated developing environment*)* som innehåller texteditor, kompilator, och andra verktyg som kan underlätta kodning och felsökning mm.

Det finns många olika sådana IDE:er. I Windowsvärlden är *Visual Studio* den mest kända, och på Mac *XCode*. I denna kurs spelar det egentligen inte någon roll vilken IDE man använder, men DMP- och DIS- studenterna måste i flera kommande kurser (DALGO, OOP) använda *Qt Creator* som finns för Windows, Mac, och Linux. Om du är DIS- eller DMP-student bör du installera *Qt Creator* redan nu!

0.2 Svenska namn men utan prickar

I sverige brukar de flesta programmerare använda engelska namn på variabler och funktioner. Även jag brukar göra detta. Men i denna kurs kommer jag faktiskt att använda svenska namn. Jag har två skäl:

1. Det blir enkelt för er att *särskilja* de saker (funktioner och variabler) som jag/vi skapat från de saker som redan ingår i C++ eller standardbiblioteket.
2. En del studenter kan vara mindre duktiga på engelska (somliga kan ha läst engelska som tredje eller fjärde språk). Även de som anser sig vara duktiga på engelska har ofta ett minst 10 gånger så stort ordförråd på svenska. Eftersom programmering kan vara nog så svårt som det är, vore det *fel att införa en ytterligare språklig svårighet* helt i onödan.

Qt Creator är en IDE som innehåller flera olika externa verktyg som kommer från olika källor. Tyvärr klarar vissa av dessa verktyg inte av att hantera svenska tecken (eller andra tecken som inte ingår i ASCII). Rekommendationen är därför att man skall skriva åääö utan prickar (dvs aao) när man namnger variabler och funktioner, etc. Endast i kodkommenterar och s.k. textsträngar bör man skriva svenska tecken.

På vissa ställen i detta kompendium kan det finns exempelkod som bryter mot denna regel och innehåller åääö, men när du skriver av koden bör du absolut ta bort prickarna (eller översätta till engelska).

(På min egen dator har jag en kompilator som faktiskt klarar av att hantera å ä ö och andra tecken som inte är ASCII, men det har visat sig att den *debugger* som *Qt Creator* använder på min dator inte är lika skicklig.)

0.3 Uppgiftstyper

De flesta programmerare gillar problemlösning. Vissa av uppgifterna är av den typen, men i många andra uppgifter skall man "bara" knappa in kod och förstå vad som händer när den exekveras/körs.

> *Hoppa inte över dessa uppgifter!* Man lär sig mycket när man skriver av kod, om man tänker samtidigt. Försök också alltid att förutse vad som kommer att hända innan du provkör en programkod som du skrivit av!

Kapitel 1

Kom igång!

I detta kapitel skall man installera en utvecklingsmiljö på sin egen dator och med hjälp av denna skapa ett program.

1.1 Installera programutvecklingsmiljön

Uppgiften i detta avsnitt är att installera programutvecklingsmiljön på den egna datorn.

Qt Creator är en utvecklingsmiljö, men det ingår egentligen inte någon kompilator i denna. Om du har en *Windows-dator* kan det vara en fördel om du först installerar Microsofts Visual Studio, för då får du en kompilator (MSVC) som *Qt Creator* kan använda. Om du har en *Mac-dator* kan det av motsvarande skäl vara en fördel om du först installerar XCode så att du får kompilatorn Clang.

(Alternativt kan du när du installerar *Qt Creator* installera kompilatorn MinGW.)

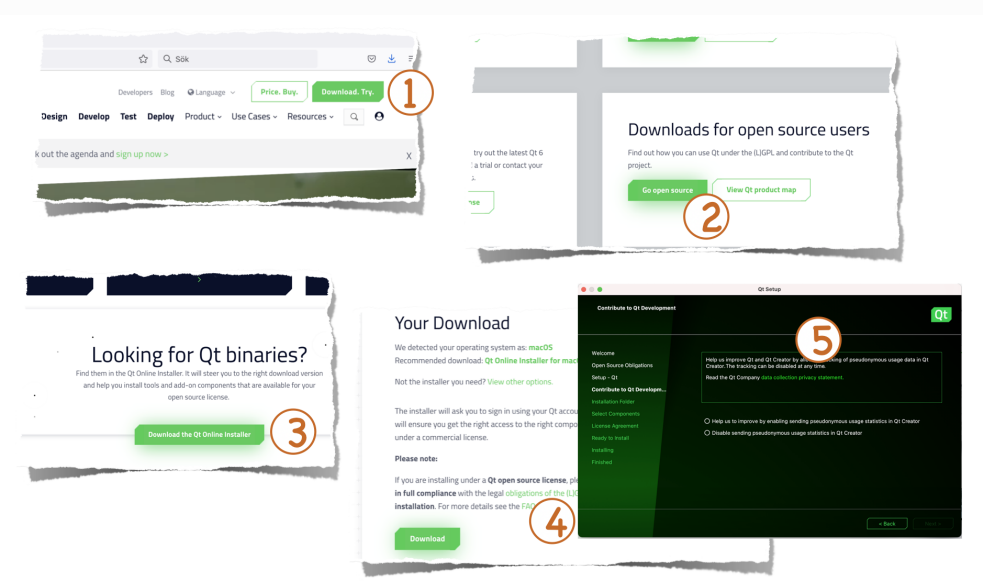
1.1.1 Installera Visual studio eller XCode

Oavsett om du tänker använda Qt Creator eller inte bör du först installera *Microsofts Visual Studio* om du har en Windowsdator och *XCode* om du har en Mac. Bägge programvarorna är gratis. Visual Studio hittar man med google och XCode via App Store.

1.1.2 Installera Qt Creator

I denna kurs kommer jag själv att använda *Qt Creator* och ibland visa er hur man använder denna IDE. Det kan därför vara praktiskt om även ni använder denna utvecklingsmiljö. Detta gäller speciellt er som läser DIS eller DMP.

Surfa till *https://www.qt.io/* och klicka dig sedan fram till den slutliga *down load knappen* för *Qt:s open source* version (se figur 1.1).



Figur 1.1: Klickanvisning för hur kan laddar ned det installationsprogram som i sin tur kummer att installera Qt och Qt Creator. För att kunna ladda ned programvaran måste man registrera sig med mejladress och lösenord.

Under installationsfasen bör du installera så få komponenter som möjligt. Jag tror inte att du behöver installera något som inte redan är förkryssat. När installationen är färdig kommer du bland annat att ha fått ett program som heter *MaintenanceTool* med vars hjälp man kan installera fler komponenter om det skulle behövas.

1.1.3 Nödlösning

Om du misslyckas med att installera *Qt Creator* på din egen dator kanske labassistenten hjälpa dig vid nästa labtillfälle.

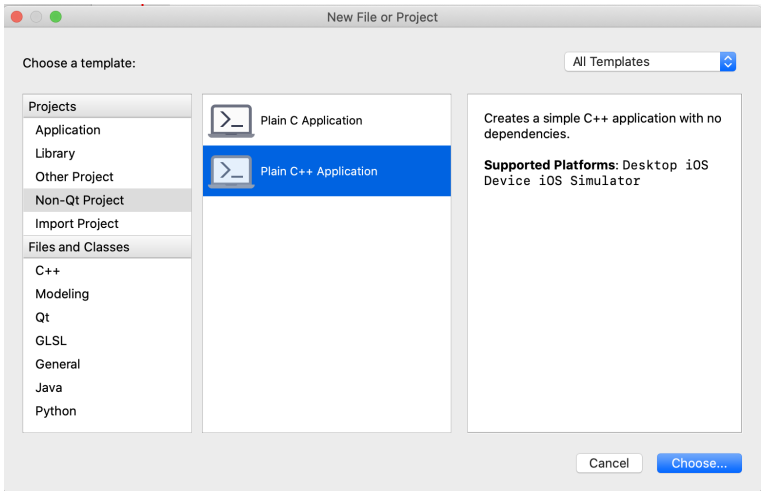
Innan dess kan du prova C++ från någon online-sida, exempelvis *http://cpp.sh*.

Du kan också köra *Qt Creator* från skolans labsalar. Eventuellt måste du först installera *Qt Creator* via *Software Center* från Windows start-meny.

1.2 Hello, world!

Här visar jag hur man skapar ett icke-grafiskt konsol/terminal projekt i Qt Creator. Gör motsvarande i den utvecklingsmiljö du använder!

Starta *Qt Creator*, och skapa sedan ett nytt projekt från fil-menyn. Välj att skapa ett "icke Qt projekt" av typ "enkel C++ applikation" enligt nedan.



När du skall välja namn på projektet kan du exempelvis välja namnet *kapitel1*.

> Varken namnet eller sökvägen får innehålla å ä ö eller andra icke-ASCII-tecken. Om du exempelvis heter *Ågren* kan det hända att sökvägen till din hemkatalog innehåller ett 'Å', vilket sannolikt leder till problem.

Qt Creator bör nu ha skapat en fil *main.cpp* med nedanstående innehåll:

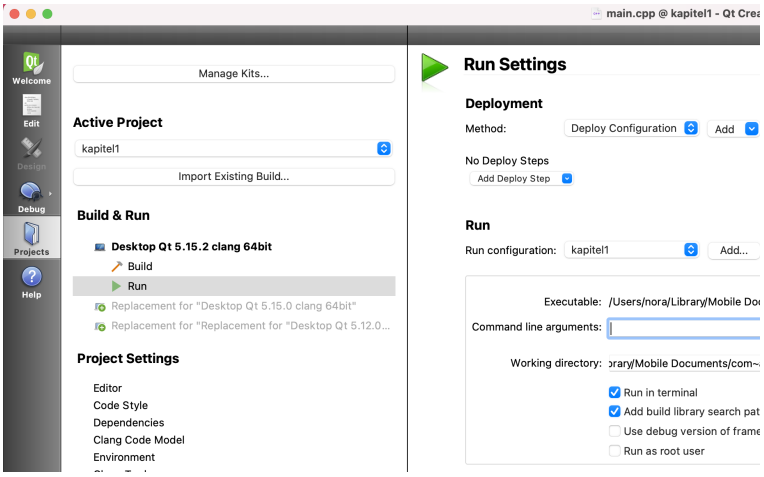
```
#include <iostream>

using namespace std;

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

Provkör programmet genom att trycka på PLAY-knappen! Programmet kommer då att *byggas* (kompileras och länkas) och sedan att exekveras (köras). Om allt fungerar som det skall kommer du att se texten

Hello world!

i ett terminalfönster (verifiera detta).

Om texten inte hamnar i ett terminalfönster, klicka på *Projects* sedan *Run* och sedan *Run in terminal* enligt nedanstående bild.



1.3 Kompileringsfel

En liten felaktighet i ett program kan ofta leda till många s.k. kompileringsfel. Och omvänt, ibland kan väldigt många kompileringsfel försvinna när man rättat ett litet fel. I denna uppgift skall du öva på detta!

Inför ett litet fel (se nedan) i föregående program. Tryck på PLAY så att koden kompileras och studera beskrivningarna av kompileringsfelen. Försök att förstå beskrivningen så gott det går. Tag sedan bort felaktigheten och verifiera att du nu kan kompilera programmet utan kompileringsfel.

Upprepa ovanstående med olika typer av små felaktigheter. Avsikten är att du skall bli bekant med felutskrifter så att du senare enklare kan hitta oavsiktliga fel.

Här följer några exempel på felaktigheter du kan prova: Ändra något tecken från gemen till versal, exempelvis *using* till *Using*, eller *int* till *Int*. Tag bort ett tecken, exempelvis ett specialtecken `# < ; { ( ) < 0`, inför en ny parentes eller krullparentes (eng. *curly braces*) på något ställe. Stoppa in ett nytt ord, exempelvis "hej" på något ställe.

Försök att generera så många kompileringsfel som möjligt genom att bara ändra ett enda tecken i källkodsfilen. Om du sitter intill en labkamrat, utmana honom/henne att hitta felet!

Kapitel 2

Funktioner, variabler och villkor

När jag skapade introduktionen till denna grundläggande programmeringskurs valde jag mellan två strategier. Antingen skulle jag börja med en snabb men ytlig introduktion till relativt många koncept, eller så skulle jag introducera betydligt färre koncept men med ett större djup. Båda strategierna har sina fördelar, men jag valde den senare då jag tror att detta bäst gagnar den absoluta nybörjaren.

Efter detta kapitel bör man förstå vad som menas med *funktioner*. Hur man använder *hel- och flyttalsvariabler*. Hur man *skriver* till terminalfönstret och *läser* från tangentbordet. Samt också *logiska uttryck*, boolska variabler, och s.k. *if-satser* (sv. vilkorssatser).

I kapitlet ingår också lite allmän *programmeringsmetodik*. Exempelvis hur man skall *namnge* funktioner och variabler, och hur man gör om man vill dela upp sitt program i olika källkodsfiler.

I den senare kursen *introduktion till programmering* kommer vi att ha *Skansholms C++ Direkt* som kursbok. Detta kapitlet motsvarar första halvan av kapitel 2 i den boken, plus funktionsanrop. Jag har också lagt in vissa övningar på *bitvisa logiska operationer* eftersom detta passar bra i den datatekniska introduktionskursen.

Börja med att skapa ett nytt projekt från Qt Creators fil-meny. Gör som du gjorde tidigare, välj ett "icke Qt projekt" av typ "enkel C++ applikation" och när du skall namnge projektet kan du exempelvis kalla det *dataIntro*, eller *introProg* om du också kommer att läsa den kursen. Jag tror att det

enklast att utföra laborationerna om man skapar *ett enda projekt* som gäller för hela detta kompendium och sedan skapar en ny fil för varje nytt kapitel (jag beskriver detta mer i detalj senare i avsnitt 2.2.9).

2.1 Ett program med utskrifter och fördröjningar

I denna uppgift skall man undersöka ett litet program som introducerar några viktiga begrepp. Man skall skriva av färdig kod, testköra och förstå den.

2.1.1 Knappa in och förstå testprogrammet

Vi skall undersöka ett testprogram som introducerar *utskrifter* och *fördröjningar*. (Jag har lagt in fördröjningarna för att visa att programmet exekveras uppifrån och ned.)

Knappa in testprogrammet

Knappa in nedanstående lilla program och tryck sedan på PLAY.

```
#include <iostream>   // innehåller cout
#include <thread>     // innehåller sleep_for
#include <chrono>     // innehåller milliseconds
#include <cmath>      // innehåller vissa mattefunktioner

using namespace std;   // Du behöver ej förstå dessa tre rader nu
using namespace this_thread;
using namespace chrono;



int main()  // Här börjar programmet
{
    sleep_for( milliseconds(500));
    cout << "Ett" << endl;
    sleep_for( milliseconds(500));
    cout << "Två" << endl;
    sleep_for( milliseconds(1000));
    cout << "Tree" << endl;
    sleep_for( milliseconds(500));

    return 0; // Detta rad behöver du ej förstå just nu.
}

```

Sannolikt får du en hel del kompileringsfel. Att rätta dessa är en del av övningen! Om du inte fick flera fel bör du lägga ned lite tid på att framkalla och rätta kompileringsfel på samma sätt som du gjorde i 1.3. När du är färdig med denna uppgift skall du kunna trycka på PLAY så att programmet körs.

Du förväntas också förstå vad programmet gör från och med den rad som innehåller ordet `main`. För att verifiera att man har förstått kan man göra små ändringar i programmet, provköra och verifiera att resultatet blev som förväntat. I detta fall kan man exempelvis ändra på fördröjningstider och utskrifter.

2.1.2 Prova autoindentering

Föregående program är vänstermarginal inte rak utan förskjuten något till höger mellan de två krullparenteserna. Detta kallas *indentering*. Kompilatorn är visserligen okänslig för hur koden är indenterad, men alla programmerare tycker att det är *superviktigt!*

I ovanstående exempel gör indenteringen att det blir lättare att lokalisera de två krullparenteserna, dvs var den s.k. main-funktionen börjar och slutar. Senare, när vi börjar skriva mer komplex kod, kommer indenteringen att hjälpa oss att få en snabb överblick över kodens struktur.

Prova auto indentering! Ändra först indenteringen så att den blir *Wild and Crazy*: Inför några extra inledande mellanslag på några rader och tag bort de inledande mellanslagen från några andra. Markera därefter hela programkoden (Ctrl-A/Command-A i Windows/Mac) och välj *Edit/Advanced/Auto Indent Selection*. Som resultat bör koden indenteras korrekt igen!

> Tips: När man letar kompileringsfel i sin kod kan det vara bra att först utföra en autoindentering, efter detta är det lättare att upptäcka felplacerade krullparenteser etc.

2.2 Funktioner

I många läroböcker introduceras *funktionsbegreppet* ganska sent. Jag anser att det är praktiskt att tidigarelägga introduktionen. Detta kommer dels

att underlätta labbandet, och därtill är det en bra programmeringsmetodik att alltid skapa funktioner som löser delproblem!

2.2.1 Skapa funktionen `räknaTillTre`

Just nu innehåller ditt program en s.k. *funktion* som heter *main*. Knappa in följande text *ovanför* denna main-funktion:

```
void raknaTillTre(){
   // TODO
}
```

Flytta nu (klipp ut och klistra in) all kod utom raden *return 0;* från main-funktionen till den nya funktionen (ersätt raden // TODO).

Om du nu provkör programmet kommer inget intressant att hända, eftersom main-funktionen är tom (sånär som på return 0 satsen).

Modifiera main-funktionen så att den ser ut enligt nedan:

```
int main(){
   raknaTillTre();
   return 0;
}
```

Om du nu provkör programmet kommer det att fungera som tidigare.

Förklaring: Från assemblerprogrammeringen bör du känna till begreppet subrutin. En *funktion* är ungefär som en subrutin. Men det finns inget nyckelord som heter CALL, istället skriver man bara namnet på funktionen följt av parenteser på det ställe där den skall anropas. När programmet körs kommer det att "hoppa" till funktionskoden och fortsätta exekveringen där, och när programkörningen når funktionens sista krullparentes (`}`) görs *automatiskt return* till den programrad som kommer efter själva funktionsanropet.

Vi kan ändra main-funktionen enligt nedan. Nu anropas `raknaTillTre` två gånger och `main` skriver dessutom själv ut lite förklarande text.

```
int main(){
   cout << "Räkna första gången: " << endl ;
   raknaTillTre();
   cout << "räkna en gång till" << endl ;
   raknaTillTre();
```



```
    return 0;
}
```

2.2.2 Återanvänd `räknaTillTre` i `räknaTillFyra`

Man kan anropa samma funktion från flera olika ställen och detta är ett sätt att *återanvända kod*. Hade det inte funnits funktioner hade man ofta varit tvungen att skriva identisk kod på flera olika ställen i sitt program, men tack vare funktionsbegreppet kan man alltså istället anropa samma funktion från dessa olika ställen. I denna uppgift skall vi återanvända `raknaTillTre` när vi implementerar `raknaTillFyra`.

Uppgift: Skriv en funktion

```
void raknaTillFyra(){
   // TODO
}
```

enligt följande:

1. Placera funktionen *mellan* `raknaTillTre` och `main` (senare förklarar jag varför detta är viktigt).
2. Låt `raknaTillFyra` anropa `raknaTillTre` innan den själv skriver ut *Fyra* på skärmen.
3. Anropa `raknaTillFyra` från `main`.

I nästa avsnitt skall vi använda lokala variabler.

2.2.3 Använd variabler i funktionen `beräknaHypotenuslängd`

Enligt *Pythagoras sats* har en rätvinklig triangel med katetlängderna x och y en hypotenusa med längden $\sqrt{x^2 + y^2}$. Knappa in nedanstående funktion strax ovanför `main`. Funktionens uppgift är att beräkna hypotenusans längd för en triangel med kateterna 3 och 4.

```
28  void beräknaHypotenuslängd(){
29     double x = 3;
30     double y = 4;
31     double xKvadrat = x*x;
32     double yKvadrat = y*y;
33     double rKvadrat = xKvadrat + yKvadrat;
34     double r = sqrt( rKvadrat );
35     cout << "x är " << x << " och y är " << y << endl;
36     cout << "Summan av kvadraterna blir " << rKvadrat << endl;
37     cout << "Vilket gör att hypotenusans längd blir " << r << endl;
38  }
```

Förklaring: Koden introducerar först variablerna `x` och `y` och initierar dem till 3 respektive 4. Ordet `double` förklaras senare[1]. Därefter introducerar vi fler variabler med lämpliga namn, exempelvis kommer `xKvadrat` att rymma värdet 9 ($x^2 = x * x = 3 * 3 = 9$). Mot slutet av funktionen beräknas `r` som kvadratroten ur `rKvadrat` (`sqrt` betecknar kvadratroten, eng *square root*). Slutligen demonstrerar funktionen också hur man med `cout` kan göra en utskrift som både innehåller textsträngar och variabelvärden.

Anropa funktionen genom att skriva `beraknaHypotenuslangd();` överst i main. Provkör!

2.2.4 Inför inparametrar till funktionen

Föregående implementering utför alltid beräkningen för en triangel med kateterna 3 och 4. Vi skall nu modifiera funktionen något. I nästa implementering inför vi s.k. *inparametrar* så att anroparen kan ange x och y:s värden. Vi passar också på att eliminera de lite onödiga variablerna `xKvadrat` och `yKvadrat`:

```
30  void beräknaHypotenuslängd(double x, double y){
31     double rKvadrat = x*x + y*y;
32     double r = sqrt( rKvadrat );
33     cout << "x är " << x << " och y är " << y << "." << endl
34          << "Summan av kvadraterna blir " << rKvadrat << endl
35          << "Vilket gör att hypotenusans längd blir " << r << endl;
36  }
```

I ovanstående kod har jag också tagit bort två `cout` satser. Tidigare hade jag tre `cout` men ovan har jag bara en, som dock sträcker sig över tre rader.

[1]Men i princip betyder det att variablerna skall lagras som ett flyttal (kommatal) i en datatyp som har dubbelt så hög precision (antal möjliga decimaler) som den enklare datatypen med namnet float har.

Detta för att demonstrera att en sats kan sträcka sig över flera rader. Observera också att de två första raderna i `cout`-satsen inte längre avslutas med semikolon. Normalt används semikolon för att avsluta satser, och här ser vi ett exempel på en sats som sträcker sig över flera rader.

Förklaring: Vi har nu angett `x` och `y` innanför de parenteser som följer funktionsnamnet. Detta gör att variablerna `x` och `y` blir s.k. *inparametrar* som anroparen kan ange.

Anropa funktionen genom att skriva nedanstående kod överst i main:

```
int main()
{
    beraknaHypotenuslangd(30,40);

    double a = 1;
    double b = 2;
    beraknaHypotenuslangd(a,b);

    double x = 2;
    double y = 3;
    beraknaHypotenuslangd(y, x); // obs: Först y sen x !!
}
```

Provkör. Beakta noga alla utskrifter och se till att du verkligen förstår allt!

Speciellt bör du ha förstått följande: I funktionen `beraknaHypotenuslangd` finns det variabler som kallas `x` och `y` och som är deklarerade som s.k. inparametrar.

Funktionen anropas 3 gånger:

1. Vid det första anropet blir `x` 30 och `y` 40.

2. Andra gången får `x` och `y` samma värden som `a` och `b` har i main, dvs 1 och 2.

3. Vid det tredje anropet blir `x = 3` och `y = 2`. Lägg märke till att variabel `x` i `main` inte är samma variabel som `x` i `beraknaHypotenuslangd`, de är två helt olika variabler som råkar ha samma namn! Variabler existerar bara lokalt i de funktioner där de deklareras.

2.2.5 Låt funktionen returnera ett värde

I nästa implementering skall funktionen få ett nytt namn. Du kan lika gärna skriva en ny funktion och låta den gamla ligga kvar. Skriv den nya funktionen strax ovanför `main`. Den nya funktionen skall ha ett *returvärde* av typ `double`. Funktionen skall returnera beräkningsresultatet till den som anropade funktionen på ungefär samma sätt som den inbyggda funktionen `sqrt` returnerar en kvadratrot till oss. Vi ger funktionen ett nytt namn eftersom funktioner som returnerar något bör ha ett namn som beskriver det som returneras!

```
41  double hypotenuslängd(double x, double y){
42      double rKvadrat = x*x + y*y;
43      double r = sqrt( rKvadrat );
44      cout << "x är " << x << " och y är " << y << "." << endl
45           << "Summan av kvadraterna blir " << rKvadrat << endl
46           << "Vilket gör att hypotenusans längd blir " << r << endl;
47      return r;
48  }
```

Anropa funktionen genom att införa nedanstående kodrader överst i main.

```
double langd = hypotenuslangd(3,4);
cout << "Funktionen returnerade längden " << langd << endl;
```

Förklaring: Det värde som funktionen returnerar hamnar i variabeln `langd`.

2.2.6 Gör funktionen *ren*

I nästa implementering gör vi så att funktionen blir *ren*. Rena funktioner har inga utskrifter eller andra sidoeffekter. De returnerar bara ett värde, och detta värde beror bara på inparametrarna.

```
double hypotenuslangd(double x, double y){
   double rKvadrat = x*x + y*y;
   double r = sqrt( rKvadrat );
   return r;
}
```

När man programmerar bör man eftersträva rena funktioner.

Rena funktioner blir dels enklare att testa (svårt att testa utskrifter) och också lättare att återanvända. Det är mer ofta man kan återanvända en funktion som bara beräknar en hypotenusa än en funktion som både beräknar hypotenusan och skriver ut densamma på skärmen.

Som programmerare bör man också eftersträva lättläst kod. Nedan ger jag därför en alternativ implementering utan onödiga variabler:

```
double hypotenuslangd(double x, double y){
   return sqrt( x*x + y*y );
}
```

2.2.7 Om namnkonventioner

Namn på funktioner och lokala variabler skall alltid ha en inledande *gemen* bokstav (dvs *lower case* bokstav, liten bokstav).

Inledande versal bokstav har man istället på egendefinierade datatyper (klasser, etc.).

Ett namn på en funktion eller variabel får inte innehålla mellanslag. Om man vill att ett namn skall innehålla flera ord måste man antingen använda kamelpuckelnotation (`raknaTillTre`) eller *undescores* (`rakna_till_tre`). Själv anser jag att att kamelpuckelnotationen har många fördelar.

2.2.8 Förstå hur funktionerna skall placeras

Efter att ha utfört denna uppgift bör du ha förstått vad som menas med en förvägsdeklaration (eng. *forward declaration*), och hur man kan undvika dessa genom att placera funktionerna i korrekt ordningsföljd.

Flytta den funktion som heter `hypotenuslangd` så att den hamnar *nedanför* main-funktionen istället för ovanför. Tryck PLAY! Nu får du kompileringsfel.

Kompilatorn läser nämligen koden uppifrån och ned, och när den hunnit fram till main-funktionens anrop till `hypotenuslangd` har den ännu inte sett att det finns en funktion som heter `hypotenuslangd`.

Man kan rätta detta fel med hjälp av en s.k. *förvägsdeklaration*. Skriv följande kodrad ovanför main:

```
double hypotenuslangd(double x, double y);
```

Kodraden anger att det finns en funktion som heter `hypotenuslangd`, och att den tar två `double`-tal som parametrar, och att den returnerar en `double`. Kodraden anger dock *inte* hur funktionen ser ut inuti! Det är bara en s.k. *deklaration* (inte en *definition*). Om du nu kompilerar bör det fungera!

Du var tvungen *deklarera* `hypotenuslangd` ovanför anropet till den, eftersom den låg *definierad* nedanför anropet. Man kan undvika förvägsdeklarationer om man placerar funktionerna i en ordning som gör att anropen alltid sker uppåt, dvs. till funktioner som kompilatorn redan har läst.

I många andra programmeringsspråk kan man placera sina funktioner i en godtycklig ordningsföljd, men som C/C++ programmerare måste man alltså tänka på att kompilatorn läser koden uppifrån och ned. Detta innebär både en fördel och en nackdel. Fördelen är att det härmed också blir enklare för oss människor att läsa koden uppifrån och ned: När vi ser ett funktionsanrop har vi redan sett den funktion som anropas och förhoppningsvis förstått vad den gör.

2.2.9 Dela upp projektet i flera cpp-filer

Efter denna uppgift bör du ha förstått hur man kan dela upp ett program i flera filer. Du bör också ha förstått hur jag tänker mig att du skall organisera din kod till övningarna i detta övningshäfte.

Jag tänker mig att du som gör dessa labbar bör *skapa ett enda projekt för hela kursen* och sedan normalt *en* (ibland kanske flera) cpp-filer per kapitel.

Från *Qt Creator*s File-meny, välj

```
new file or project... / C++ / C++ source file
```

Välj ett lämpligt namn på filen som inte innehåller åäö, exempelvis:

kap2FunkVarVillkor.cpp

se till att den hamnar i rätt filkatalog och att den adderas till rätt pro-fil.

Kopiera all kod (inklusive main-funktionen) från *main.cpp* till den nya filen. Byt sedan namn och returvärde på den på den nya main-funktionen från

```
int main()
```

till

```
void ingangTillKap2FunkVarVillkor()
```

och tag också bort den `return 0;` sats som står längst ned i funktionen.

En fil kan innehålla många funktioner som anropas från andra filer och man behöver inte namnge dem på något speciellt sätt. För att underlätta labbandet *i denna kurs* tänker jag mig dock att varje fil som skapas normalt endast skall ha en funktion som anropas utifrån, och att denna funktion skall heta

ingangTill*AktuelltFilnamn*

För att funktionen skall anropas måste du återgå till main-filen och ändra den enligt nedan.

```
1  #include <iostream>
2
3  using namespace std;
4
5  void ingångTillKap2FunkVarVillkor();
6
7  int main(){
8      cout << "Hello world!" << endl;
9
10     ingångTillKap2FunkVarVillkor();
11     return 0;
12 }
```

(Om du följer mitt råd att undvika svenska tecken måste byta *å* mot *a* och namna funktionen *inganTill...* istället, se avsnitt 0.2.)

På rad 5 förvägsdeklarerar vi den funktion som vi vill anropa, och på rad 10 anropar vi den. Rad 1 och 3 behövs eftersom vi på rad 8 skriver ut något på skärmen. Alla program har en main-funktion som anropas av operativsystemet när det startar programmet och programmet avslutas när `main` har kört klart. Operativsystemet vill då veta om programmet lyckades med sin uppgift eller inte. Main-funktioner skall därför returnera en felkod. Vi returnerar 0 för att ange att inget fel inträffat.

I varje kapitel kommer du att skriva diverse kod. Jag rekommenderar dig att du skapar en cpp-fil för varje kapitel (blir den väldigt lång kan du skapa flera) och placerar kapitlets kod i små funktioner som du anropar från den aktuella filens `ingangTill`-funktion. De små funktionerna bör placeras ovanför `ingangTill`-funktionen, så att du slipper *forward*-deklarera dem. När du skrivit en ny funktion bör du placera anropet till den *överst* i `ingangTill`-funktionen, så att den funktion du skrev senast anropas först då du kör programmet!

Generella labanvisningar

I detta kompendium är normalt varje kapitel lagom omfattande för att motsvara en egen cpp-fil.

1. Varje gång du påbörjar nytt kapitel bör du skapa en ny fil med passande namn (som dock inte får innehålla åääö eller mellanslag). Exempelvis *kap2FunkVarVillkor.cpp*.
2. Inför en funktion `ingangTill`*Filnamn* längst ned i den nya filen.
3. I main-filen, *forward*-deklarera den nya funktionen ovanför `main` och anropa den överst i main.
4. När du skall skriva testkod: Placera koden i en ny funktion i den nya filen, och anropa den överst i `ingangTill`-funktionen.

2.3 Flyt- och heltalsvariabler

Efter att ha läst detta avsnitt skall du förstå vad som menas med tilldelning. Du skall förstå hur flyttal och heltal fungerar, och du skall ha löst några enklare programmeringsuppgifter.

2.3.1 Prova *tilldelning*

Beakta nedanstående programsnutt (knappa gärna in den i en liten funktion i den aktuella filen och provkör):

```
double x = 10;
double y = 20;
x = y;
```



```
cout << "x är " << x << " och y är " << y << endl;
y = 100;
cout << "x är " << x << " och y är " << y << endl;
```

Ovan introducerar vi först två variabler (`x,y`) och *initierar* dem med värdena 10 respektive 20. Nästa sats lyder: `x = y`. Den som kan lite matematik men inte programmerat tidigare måste tycka att detta är konstigt (för att inte säga felaktigt). Men likamedtecknet betyder *inte* samma sak som i matematiken. Här betyder det *tilldelning*. Variabeln `x` tilldelas det värde som `y` har. Värdet 20 kopieras från `y` till `x`. (Tilldelningen sker alltid från höger till vänster.) Sedan skriver vi ut variabelvärdena. I satsen som följer tilldelas `y` värdet 100. Den som är van vid *Excel*-programmering kan luras att tro att även `x` får värdet 100, men så är icke fallet. Variabeln `x` kommer fortfarande att ha värdet 20.

Att öka med 1

Låt säga att man vill ge en variabel ett nytt värde som beror av dess tidigare värde, exempelvis att man vill öka värdet med 1. Detta kan göras på flera olika sätt. Exempelvis kan man skriva:

```
x = x + 1;
```

Låt säga att `x` från början har värdet 10. När tilldelningen skall utföras kommer datorn först att beräkna högerledet till 11. Därefter kommer vänsterledet att tilldelas detta värde.

Alternativt kan man använda följande förkortade skrivsätt:

```
x += 1;
```

Detta skrivsätt fungerar också med andra räknesätt och andra konstanter. Exempelvis betyder `x *= 2` att `x` skall multipliceras med 2.

I det fall man vill öka ett värde med just 1 kan man också använda något av nedanstående lite ålderdomliga skrivsätt:

```
++x;
x++;
```

För att förstå skillnaden kan du provköra följande kod:

```
int x = 10;
int  resultat = ++x;
cout << "resultat = " << resultat << " x = " << x << endl;
x = 10;
resultat = x++;
cout << "resultat = " << resultat << " x = " << x < endl;
```

Om man tänker sig att `++x` skulle utföras av en funktion så skulle den först öka `x` och därefter returnera resultatet (pre-inkrement). Funktionen som utför `x++` är lite krångligare. Den sparar först `x`:s ursprungliga värde, därefter ökar den `x` och slutligen returnerar den det ursprungliga sparade värdet (post-inkrement).

Personligen brukar jag använda skrivsättet `x+=1` i de fall jag bara vill öka en variabel med 1 (och inte bryr mig om att lagra resultatet i någon annan variabel).

2.3.2 Lär dig inläsning och utskrift av flyttal

Innan du utför nästa programmeringsuppgift måste du veta hur man läser in ett flyttal som användaren knappar in. I nedanstående programsnutt ber jag användaren att knappa in ett flyttal. Därefter skriver jag ut det på skärmen, och därefter skriver jag igen ut det på skärmen men nu avrundat till 2 värdesiffrors noggrannhet. För att få den sista satsen att fungera (dvs `setprecision(2)`) måste man ha inkluderat *<iomanip>*. Skriv därför

```
#include <iomanip>
```

högt uppe i programmet (ovanför alla funktionsdefinitioner).

```
void provaInlasningOchUtskrift(){
    cout << "Knappa in ett decimaltal:";
    double decimaltal = 0;
    cin >> decimaltal;
    cout << "Du knappade in talet " << decimaltal << endl;
    cout << "Med två värdesiffror: " << setprecision(2) << decimaltal  << endl;
}
```

`cin` representerar alltså tangentbordet. Satsen `cin >> decimaltal` gör att det data (decimaltal) som användaren kappar in hamnar i variabeln `decimaltal`.

2.3.3 Programmera `beräknaBokkostnad`

Ibland kan det kännas dyrt att köpa en lärobok för flera hundra kronor, men om man betänker hur lång tid det tar att läsa boken blir priset per timme ofta inte särskilt högt. Nedan skall du skriva ett program som beräknar detta. Fråga först vad boken kostar. Fråga sedan hur många sidor boken har. Fråga sedan hur stor procentandel av boken som användaren kommer att läsa. Fråga hur många minuter det tar att läsa en sida. Skriv därefter ut vad boken kostar per timme. Detta tal skall skrivas med två decimaler.

Exempel på programkörning (de feta talen knappas in av användaren):

> Hur många kronor kostar boken? **513**
> Hur många sidor har boken? **657**
> Hur många procent av sidorna kommer du att läsa? **50**
> Hur många minuter tar det att läsa och förstå en sida? **15**
> (Tystnad, beräkning pågår)
> Du beräknas ägna boken ca 82.1 timmar.
> Boken beräknas därför kosta dig 6.25 kr/timme.

Tips:

1. Inför lämpliga variabler med lättbegripliga namn (`bokPris`, `antalSidor`, `procentAttLasa`, `minuterPerSida`) alla av typen `double`.
2. Läs in deras värden med hjälp av `cin` (skriv först ut frågan med `cout`).
3. Skriv på lämpligt ställe ut ut *"(Tystnad beräkning pågår)"* och låt också programmet pausa under ca 3s. Detta förhöjer användarupplevelsen och får beräkningsresultatet att verka mer trovärdigt ;-).
4. Inför variabeln `totaltAntalTimmar` och initiera den med sitt korrekta värde som du beräknar på samma rad.
5. Inför variabeln `kostnadPerTimme` och initiera även den med sitt korrekta värde.
6. Skriv slutligen ut vad boken beräknas kosta per timme med två decimalers noggrannhet (se avsnitt 2.3.2).

2.3.4 Testa flyttalens begränsningar

Vi skall nu dyka lite djupare i datatypen `double`.

I vissa mycket enkla (gamla) CPU:er har alla decimaltal (kommatal) ett fixt antal decimaler, exempelvis två. Att räkna med dessa s.k. fix-tal är nästan som att räkna hundradelar med heltal. Decimaltal med variabelt antal decimaler brukar kallas *flyttal* eftersom decimalpunkten inte är fix utan flytande. I C/C++ finns datatyperna `float` och `double` för att representera flyttal. Ett flyttal av typ `double` har dubbelt så många värdesiffror som en `float`.

Internt i datorn lagras ett flyttal av typ `double` ofta i 64 bit. Detta gör att såväl precision (antal värdesiffror) som talområdet blir mycket stort, och att flyttal därför beter sig ungefär som man kan förvänta sig. Nedanstående övning avser att visa de begränsningar som flyttal trots allt har.

Knappa in och provkör nedanstående experimentkod! Du kommer att lära dig att 0.3 och 3×0.1 inte lagras som exakt samma tal. Vidare kommer du att se vad som händer om man skulle råka dividera med 0. Det visar sig att ett flyttal också kan representera vissa ogiltiga tal (infinity, -infinity, not-a-number).

```
void testaFlyttal(){
    double nollKommaEtt = 0.1;
    double nollKommaTreA = 0.3;
    double nollKommaTreB = 3*nollKommaEtt;

    cout << endl;
    cout << "Skriver ut 0.1 och 0.3 och 0.3:" << endl;
    cout << nollKommaEtt  << endl;
    cout << nollKommaTreA << endl;
    cout << nollKommaTreB << endl;

    cout << setprecision(17) ;
    cout << endl;
    cout << "Skriver ut 0.1 och 0.3 och 0.3 igen:" << endl;
    cout << nollKommaEtt << endl;
    cout << nollKommaTreA << endl;
    cout << nollKommaTreB << endl;

    double noll = 0;
    double två = 2;

    cout << endl;
    cout << "Division med noll:" << endl;
    cout <<  noll/noll <<endl;
    cout <<  två/noll <<endl;
    cout <<  -två/noll <<endl;

    cout << endl;
    cout << "En liten lek med oändligheten:" << endl;
    double oo = två/noll;
    cout << två * oo << endl;
    cout << oo + oo << endl;
    cout << oo - oo << endl;
    cout << -oo - oo << endl;

    cout << endl;
    cout << "Division med oändligheten:" << endl;
    cout << två/oo << endl;
    cout << -två/oo << endl;
    cout << oo/oo << endl;
    cout << -oo/oo << endl;
    cout << endl;
}
```

2.3.5 Prova heltalsdivision

Datatypen `int` används för att representera heltal (eng *integer*).

Knappa in och provkör nedanstående experimentkod. Koden inleds med ett experiment med mycket stora heltal. Den fördefinierade konstanten `INT_MAX` innehåller det största heltal som en `int` kan lagra, och koden

försöker att skapa ett ännu större heltal. *Vad händer, och varför?* Den resterande experimentkoden behandlar division. Kvoten mellan två heltal blir alltid ett heltal, den kan aldrig bli `0.4` eller `inf`, etc., dessutom sker inte avrundning utan trunkering (decimalerna försvinner). Försök att förutsäga resultatet innan du provkör och testar om du hade rätt.

```
100 void testaHeltal(){
101     int störstaHeltalet = INT_MAX;
102     int ännuStörre = störstaHeltalet + 1;
103 
104     cout << endl;
105     cout << "Stora heltal:" << endl;
106     cout << "störstaHeltalet: " <<   störstaHeltalet <<  endl;
107     cout << "ännuStörre:      " <<   ännuStörre <<  endl;
108 
109 
110     int noll = 0;
111     int två = 2;
112     int fem = 5;
113 
114     cout << endl;
115     cout << "Några heltalsdivisioner:" << endl;
116     cout <<  fem/två <<endl;
117     cout <<  två/fem <<endl;
118     cout <<  två/fem + två/fem + två/fem <<endl;
119 
120     cout << endl;
121     cout << "Division med noll:" << endl;
122     cout <<  noll/noll <<endl;
123     cout <<  två/noll <<endl;
124 }
```

När beräkningen leder fram till ett tal som inte kan representeras som ett heltal måste programmet krascha, vad skulle det annars göra?

2.3.6 Lär dig använda hel- och flyttalslitteraler

Det finns två sätt att ange ett tal. Antingen anger namnet på en variabel som innehåller det sagda talet eller så anger man talet med siffror som en s.k. *litteral*. Om man ger talet som en litteral kommer kompilatorn att anse att det är ett heltal om det ges utan decimalpunkt.

Försök att förutsäga resultatet av nedanstående programkörning, och testa sedan om du hade rätt!

```
126 void testaLitteraler(){
127     cout << endl;
128     cout << "testar beräkningar med literaler" << endl;
129     cout << 7/10*500.0 << endl;
130     cout << 7/10.0*500 << endl;
131     cout << 3/4 + 3/4 + 3/4 + 3/4  << endl;
132     cout << 3.0/4 + 3/4 << endl;
133 }
```

2.3.7 Programmera växelpengar (version 1)

Låt säga att man handlar något, betalar kontant, och skall ha tillbaka växel. Skriv en funktion som hjälper personen i kassan att ge tillbaka rätt antal sedlar och mynt.

```
void skrivUtVaxel(int antalKronor){
     // TODO
}
```

Vi kan anta att det finns sedlar och mynt av följande valörer: 500, 200, 100, 20, 10, 5, 2 , 1. Om man anropar funktionen `skrivUtVaxel( 947 )` skall den skriva ut följande:

```
947 kronor är:
   1 x 500
   2 x 200
   0 x 100
   2 x 20
   0 x 10
   1 x 5
   1 x 2
   0 x 1
```

Krångla inte till beräkningarna med för mycket matematik utan skriv en funktion som är enkel att förstå! Här är de tre första raderna i en möjlig implementering:

```
int kronorKvarAttBetala = antalKronor;
int antal500 = kronorKvarAttBetala/500;
kronorKvarAttBetala -= antal500*500;
...
```

Observera att ovanstående kodrader drar nytta av att heltalsdivision inte ger några decimaler!

2.3.8 Lär dig moduloberäkningar

Om a och b är två heltal är a mod b den rest man får efter heltalsdivision. Exempelvis är 123 mod 10 = 3, eftersom

$$\frac{123}{10} = 12 + \frac{3}{10} \tag{2.1}$$

I C/C++ utförs moduloberäkningen med `%`-operatorn. `123 % 10` blir med andra ord 3.

Försök att förutsäga resultatet av nedanstående programkörning, och testa sedan om du hade rätt!

```
93  void testaModulo(){
94      cout << "testar moduloberäkningar" << endl;
95
96      cout << 13 % 10 << endl;
97      cout << 57 % 10 << endl;
98      cout << 7 % 5 << endl;
99      cout << -57 % 10 << endl;
100 }
```

Observera att resultatet av en moduloberäkning kan bli negativt!
Kontrollfråga: vad borde `-58 % 10` bli?

2.4 Heltal som bitmönster

I vissa tillämpningar vill man manipulera heltal på bit-nivå. I dessa tillämpningar är man ofta mindre intresserad av heltalens heltalsvärden och mer intresserad av deras bitmönster, dvs vilka bitar som är 0 respektive 1. I dessa tillämpningar brukar man ofta använda `unsigned int` och inte sällan anger man också heltalen i hexadecimal bas.

I nedanstående exempel initierar vi heltalet `v` med litteralen `0xa7`. Prefixet `0x` gör att kompilatorn förstår[2] att vi avser att ge en heltalsliteral i basen 16 (hex). Heltalet får alltså värdet 167 eftersom $a7_{16} = 167_{10}$.

```
unsigned int v = 0xa7;
```

[2]En numerisk litteral måste börja med en siffra och en variabel får inte göra det. Hade vi bara skrivit `a7` hade kompilatorn förväntat sig att det fanns en variabel med detta namn, och genererat kompileringsfel om så inte var fallet.

2.4.1 Lär dig bitvisa operationer

På heltal kan man också använda bitvisa logiska operationer, exempelvis är operatorn `&` bitvis *och* (eng. *AND*). För att förstå hur den fungerar måste man tänka på hur talen lagras binärt. Nedanstående uträkning visar varför `6 & 3` blir 2:

6	=	0	0	1	1	0
3	=	0	0	0	1	1
6 & 3	=	0	0	0	1	0

Andra bitvisa logiska operationer är *eller* (`|`), *xor* (`^`), och *not* (`~`).

Man kan också *skifta* alla bitar i ett heltal ett visst antal steg till vänster (`<<`) eller till höger (`>>`). Nedan ser vi några exempel:

1 << 4	=	0...000001 << 4	=	0...010000	=	16
7 << 2	=	0...000111 << 2	=	0...011100	=	28
28 >> 2	=	0...011100 >> 2	=	0...000111	=	7

Som bekant används `<<` operatorn också vid utskrifter, vilket gör att nedanstående kodrad helt enkelt skriver ut `72` i terminalfönstret.

```
cout << 7 << 2 << endl;
```

För att på ett enkelt sätt testa skift-operatorn kan man använda parenteser. Nedanstående kodrad

```
cout <<  (7 << 2) << endl;
```

beräknar först `7<<2` och skriver därefter ut resultatet (dvs 28) i terminalfönstret.

2.4.2 Programmera färginformation som `unsigned int`

Ett mänskligt öga innehåller receptorer för *rött-, grönt-* och *blått* ljus. En digital bild lagrar därför en *röd, grön* och *blå* ljusintensitet för varje bildpunkt. Vissa bildformat tillåter dessutom transparens, då lagras också information om hur pass ogenomskinlig (eng *opaque*) varje bildpunkt är.

Ovan beskrivna färginformation lagras ofta i en 32 bit `unsigned int` som skall tolkas som 4 *byte*-värden mellan 0 och 255. Se nedan:

$$\underbrace{01010011}_{r=83}\underbrace{11111100}_{g=252}\underbrace{10110100}_{b=180}\underbrace{01111111}_{opacity=127}$$

Implementera färginfo

Implementera funktionen `farginfo` vars uppgift är att returnera en `unsigned int` som innehåller den angivna färginformationen som 4 *byte*-värden.

```
unsigned int farginfo(unsigned int röd, unsigned int grön, unsigned int blå,
                                                    unsigned int ogenomskinlighet){
   // TODO
}
```

Tips: Använd *vänsterskiftningar* och bitvisa *eller*!

Implementera extraheringsfunktionerna

Implementera funktionen `blaFranFarginfo` vars uppgift är att extrahera den blå intensiteten från inparametern och returnera denna:

```
unsigned int blaFranFarginfo(unsigned int farginfo){
   // TODO
}
```

Tips: Högerskift och *och*.

Implementera också `gronFranFarginfo, rodFranFarginfo`, och `ogenomskinlighetFranFarginfo` med uppenbara uppgifter.

Testa också dina funktioner med hjälp av nedanstående testkod

```
unsigned int farg = farginfo(10,20,30,40);
cout << rodFranfarginfo( farg ) << endl;
cout << gronFranFarginfo( farg ) << endl;
cout << blaFrånfarginfo( farg ) << endl;
cout << ogensomskinlighetFranfarginfo( farg ) << endl;
```

och verifiera att den skriver ut `10, 20, 30` och `40`.

2.4.3 Programmera `talFranDecimalDotNotation`

(Denna uppgift är mycket lik en av de tidigare uppgifterna)

Inom nätverkstekniken anger man ofta 32 bitars IPv4-adresser med en så kallad *dotted decimal* notation. De 32 bitarna delas upp i 4 grupper om 8 bitar i varje, och varje sådan grupp skrivs sedan som ett heltal mellan 0 och 255 med punkter mellan talen.

Exempelvis skrivs den binära talet 01010011111111001011010000001111 som 83.252.180.15, eftersom:

$$\underbrace{01010011}_{83}\underbrace{11111100}_{252}\underbrace{10110100}_{180}\underbrace{00001111}_{15}$$

Uppgift: Implementera nedanstående funktion vars uppgift är att returnera det 32 bitars heltal som motsvarar *tal3.tal2.tal1.tal0* i DDN *(dotted decimal notation)*.

```
unsigned int talFranDecimalDotNotation(int tal3, int tal2, int tal1, int tal0){
   // TODO
}
```

Samtliga fyra inparametrar antas vara tal mellan 0 och 255.

2.4.4 Programmera `nätmask`

Inom nätverkstekniken behöver man ibland använda ett tal bestående av `n` 1:or följd av `32-n` 0:or, exempel:

$$\underbrace{111111111111}_{12}\underbrace{00000000000000000000}_{20}$$

Uppgift: Implementera nedanstående funktion vars uppgift är att returnera det 32 bitars heltal som inleds med n 1:or och följs av 32-n 0:or.

```
unsigned int natmask(int antalInledandeEttor){
   // TODO
}
```

(Om man vill nollställa de `k` högraste bitarna i ett heltal kan man skifta bitarna åt höger `k` steg och sedan tillbaka åt vänster `k` steg.)

2.5 Booleska variabler, logiska villkor och if-satsen

Efter detta labbavsnitt bör du förstå booleska variabler, loogiska uttryck, och if-satser.

En variabel av typen `bool` kan bara anta värdena `true` och `false` (eng. för sant och falskt). Nedan ser vi lite kod som använder `bool`-variabler:

```
bool gillarHundar = true;
bool gillarSpindlar = false;
double kroppstemperatur = 37.2;
bool harFeber = (kroppstemperatur>=38);
```

Man bör eftersträva att ge de booleska variablerna *namn* som beskriver vad som gäller när de är sanna.

Om man jämför två tal med hjälp av någon av de 6 jämförelseoperatorerna[3] blir resultatet av jämförelsen ett *booleskt värde*, i ovanstående exempel får därför `harFeber` värdet `false`.

Med hjälp av logiska operartorer kan man kombinera flera booleska värden.

```
bool ärNormal = gillarHundar && !gillarSpindlar;
```

De logiska operatorerna är för övrigt: `&&` (och), `||` (eller), `!` (icke).

2.5.1 Lär dig hur boolska uttryck beräknas

I detta avsnitt skall vi fördjupa oss i följande påstående:

> I programspråket C/C++ beräknas logiska uttryck från vänster till höger och inga uppenbart onödiga beräkningar görs.

Antag att du skrivit kod som skall beräkna

```
a && b
```

där `a` och `b` är två boolska uttryck som kanske också behöver beräknas. Uttrycket kommer att beräknas från vänster till höger. Antag att `a` blir

[3] jämförelseoperatorer: == , != , < , <= , > och >=

`false`, då kommer garanterat `a && b` att bli `false` oberoende av `b` (*eller hur?*). Om `a` är `false` är det med andra ord onödigt att beräkna `b`, och programspråket garanterar att så icke sker. Om du istället skrivit kod som skall beräkna `c || d` kommer `d` inte att beräknas om `c` är `true`.

Nedanstående kod definierar först 4 funktioner (på ett ganska fult sätt med en rad/funktion). Funktionerna heter t1,t2,f1 och f2, där t-funktionerna returnerar `true` och f-funktionerna `false`. Varje funktion skriver också ut sitt eget funktionsnamn, så att man kan se när de blir anropade.

I den testfunktion som följer anropas dessa funktioner. Försök att förutse vilken utskrift som genereras när testkoden körs! Knappa sedan in koden och provkör.

```
bool t1(){ cout << "t1 " ;  return true;  }
bool t2(){ cout << "t2 " ;  return true;  }
bool f1(){ cout << "f1 " ;  return false;  }
bool f2(){ cout << "f2 " ;  return false;  }

void testaBoolUttryck(){
    cout << "Testar boolska uttryck" << endl;

    bool b;
    b = t1() && (t2() || f1());
    cout << "resultat " << b << endl;
    b =  t1() || t2() || f1();
    cout << "resultat " << b << endl;
    b =  f1() || (t2() && (f1() || f2()));
    cout << "resultat " << b << endl;
}
```

Lär dig hur beräkningsordningen kan utnyttjas

Antag att vi vill undersöka om heltalet `a` är en multipel av heltalet `b`. Nedan ser vi ett försök att göra detta med hjälp av modulooperatorn:

```
int a = ....
int b = ....
bool arMultipel = (a % b == 0);
```

Ovanstående kod är dock *farlig*, eftersom den får programmet att krascha om `b` är 0. Nedan ser vi två försök att lösa detta problem:

```
bool arMultipel = (a % b == 0) && (b!=0);  // lösningsförsök A
bool arMultipel = (b!=0) && (a % b == 0);  // lösningsförsök B
```

Vilken av ovanstående lösningar är ok? Provkör om du är osäker!

2.5.2 Knappa in funktionen `ungefärLika`

Självklart kan man även skriva meningsfulla funktioner som returnerar booleska värden. Knappa in nedanstående funktion:

```
bool ungefarLika(double a, double b){
   double epsilon = 0.001;
   double diff = a-b;
   return  (diff < epsilon) && (diff  > (-epsilon));
}
```

Vad gör den? Provkör om du är osäker!

2.5.3 Lär dig använda if-sats

Med `if`-satsen kan man exekvera kod villkorligt. En if-sats består av ett boolskt uttryck, som måste skrivas innanför parenteser, och en kodsats. Kodsatsen skall endast exekveras *om* (eng *if*) det boolska uttrycket är sant.

`if (`*`booleskt uttryck`*`)`
  kodsats;

Inte sällan är kodsatsen ett *kodblock* bestående av flera kodsatser innanför krullparenteser.

Knappa in nedanstående kodexempel med två `if`-satser. I den ena if-satsen har jag gett ett kodblock som skall köras om villkoret är sant.

```
double a = 0.3;
double b = 0.1 + 0.1 + 0.1;
if ( ungefarLika(a,b) ){
    cout << "a och b har ungefär samma värden." << endl;
    if (a == b)
        cout << "De har faktiskt exakt samma värden!" << endl;
    else
        cout << "Men bara ungefär." << endl;
}
else cout << "a och b är inte särskilt lika" << endl;
```

1. Vad gör koden?
2. Vilken utskrift genererar den?
3. Kompilatorn bör varna på en av kodraderna. På vilken rad, och vad varnar den för?

2.5.4 Förstå sambandet mellan `int` och `bool`

I programspråket C (och i äldre versioner av C++) finns inte datatypen `bool`. Istället används `int`. I C är 0 *falskt* och alla andra heltal är *sanna*. Eftersom programspråket C++ har ambitionen att vara bakåt kompatibel med C skall kod som fungerar i C helst också fungera i C++. I C++ finns därför en *automatisk typkonverteringen* från `int` till `bool`. Talet 0 konverteras till `false` och alla andra tal till `true`. Typkonverteringen utförs när kompilatorn förväntar sig att läsa en `bool` men istället hittar en `int`. Nedan ser vi ett exempel på detta. Om du provkör ser du att bara den första raden skrivs ut.

```
if (5)
   cout << "Denna rad skrivs ut" << endl;
if (0)
   cout << "Skrivs inte ut" << endl;
```

Typkonverteringen utnyttjas ibland av programmerare som eftersträvar kort kod. Nedanstående två `if`-satser är ett exempel på detta. Satserna gör samma sak.

```
if (antalFrukter != 0)
   cout << "Jag ser att du har köpt frukt!" << endl;
if (antalFrukter)
   cout << "Jag ser att du har köpt frukt!" << endl;
```

Även om jag själv brukar eftersträva kort kod föredrar jag faktiskt det första skrivsättet, eftersom det låter bäst när det högläses och eftersom jag programmerar i C++ och inte i C.

Ett vanligt skrivfel

I nedanstående kodexempel utnyttjar vi igen den automatiska typkonverteringen från `int` till `bool`. Men i två av de boolska uttrycken har vi av misstag skrivit `&` istället för `&&`. Detta orsakar en bug som blir svårupptäckt eftersom den bara yttrar sig ibland.

```
int antalÄpplen = 4;
int antalMeloner = 1;
if (antalÄpplen && antalMeloner)
   cout << "Jag ser att du har köpt olika slags frukter!" << endl;
if (antalÄpplen & antalMeloner)  // BUG
   cout << "Jag ser att du har köpt olika slags frukter!" << endl;
```

```
antalÄpplen = 5;
if (antalÄpplen & antalMeloner)  // BUG
   cout << "Jag ser att du har köpt olika slags frukter!" << endl;
```

Förklara varför endast två av ovanstående 3 utskrifter utförs!

Ett annat vanligt skrivfel

Här ser vi ytterligare ett exempel på hur den automatiska konverteringen från `int` till `bool` kan skapa problem. *Klistra in och provkör* nedanstående kod (som innehåller ett relativt svårupptäckt skrivfel):

```
double summa = 125.0;
int antal = 10;

if (antal = 0)
    cout << "medelvärdet kan ej beräknas" << endl;
else {
    double medelvarde = summa/antal;
    cout << "medelvärdet är " << medelvarde <<  endl;
}
```

Man skulle kunna tro att kodsnutten skrev ut att *medelvärdet är 12.5*, men så kommer icke att ske. Vad händer, och varför? På vilket sätt har typkonverteringen från `int` till `bool` ställt till det för oss?

(om du suttit några timmar och fortfarande inte kan rätta buggen bör du gå vidare till nästa avsnitt.)

Lösningen kan stavas "const"

Varje gång man inför en variabel som man ej avser att ändra bör man tydliggöra detta med hjälp av nyckelordet `const`. Ändra de två första raderna i ovanstående kodsnutt enligt:

```
const double summa = 125.0;
const int antal = 10;
```

Detta har två fördelar:

1. Det blir lättare att förstå koden om man omedelbart förstår att `summa` och `antal` är konstanter som inte kommer att ändras av den kod som följer.

2. Kompilatorn genererar kompileringsfel om man av misstag skulle skriva kod som ändrar en konstant.

Om du kompilerar den modifierade koden kommer du nu att få ett kompileringsfel på den kodrad som orsakade problemet!

Pga lättja brukar de flesta programmerare (inklusive jag själv) inte använda nyckelordet `const` så ofta som vi borde.

Kapitel 3

Loopar

Detta kapitel innehåller övningar som tränar på loopar (`while`- och `for`). Skapa först en ny labfil och kalla den exempelvis *kapitel3.cpp*. (se de generella anvisningarna i 2.2.9.)

3.1 Kylskåpsloopen

Tidigare har du troligtvis skapat ett assemblerprogram som tänder och släcker en varningslampa på ett kylskåp. Här skall du implementera motsvarande kod med en s.k. `while`-loop. För varje varv i loopen skall användaren knappa in en temperatur, och datorn skall sen skriva ut om det är för kallt, för varmt, eller lagom. Loopen skall avbrytas om användaren knappar en en temperatur som är lägre än den absoluta nollpunkten. Uppgiften går ut på att implementera detta på två olika sätt, med och utan `break`.

3.1.1 kylskåpMedWhileTrue

En s.k. `while (true)`-loop loopar i all evighet ända tills den avbryts med `break` (eller `return`). Här skall vi öva på denna konstruktion.

Implementera nedanstående funktion:

```
void kylskapMedWhileTrue(){
   cout << "Välkommen till kylskåpsloopen 1" << endl;
   while (true){
```

```
        // TODO
    }
    cout << "Lämnar kylskåpsloopen 1" << endl;

}
```

Ersätt TODO-raden med kod som gör följande:

1. Deklarera temperaturvariabel och läs in temperaturen som ett decimaltal från användaren.

2. Om användaren ger en temperatur som är mindre än den absoluta nollpunkten $-273°$ skall funktionen *inte* skriva att det är för kallt utan istället avbryta loopen med `break`.

3. Annars, om temperaturen $\leq 2°$ skall funktionen skriva *för kallt*, om temperaturen $\geq 8°$ *för varmt* och om det varken är för varmt eller för kallt skall den skriva *lagom*. Därefter skall funktionen loopa tillbaka och upprepa frågan till användaren.

3.1.2 kylskapMedWhileVillkor

Implementera nedanstående funktion

```
void kylskapMedWhileVillkor(){
    cout << "Välkommen till kylskåpsloopen 2" << endl;
    double temperatur = 0;
    while (temperatur > -273){
        // TODO
    }
    cout << "Lämnar kylskåpsloopen 2" << endl;
}
```

Ersätt TODO med lämplig kod som gör så att funktionen beter sig på samma sätt som föregående funktion utan att använda `break`. Tänk på att inte skriva att det är för kallt när användaren vill avbryta loopen. Observera också att vi nu är tvungna att deklarera `temperatur` utanför while-loopen.

3.2 Gissa tal

I den aktuella labfilen skall du implementera följande lek/spel-program: Datorn skall "tänka" på ett tal mellan exempelvis 0 och 100, och användarens uppgift är att gissa vilket talet är. Efter varje gissning skall datorn ange om användaren gett ett för litet, för stort, eller korrekt tal. Exempel på programkörning:

> Datorn tänker på ett tal mellan noll och 100. Gissa vilket!
> din gissning: **25**
> För litet.
> din gissning: **80**
> För stort.
> din gissning: **31**
> Vilken tur!
> Du fick rätt på 3:e försöket!

Nedan visar jag hur man kan strukturera upp programmet i funktioner. För att få tillgång till den `rand`-funktion som används måste du inkludera `<stdlib.h>` högt uppe i cpp-filen. Den färdigskrivna funktionen `nyttTal(n)` returnerar ett slumptal mellan 0 och n (se avsnitt **??**). Funktionen `anvandarensGissningGissning` bör skriva ut texten *din gissning:* på skärmen, och därefter läsa in och returnera det heltal som användaren knappar in. Funktionen `utforEnSpelomgang` är halvfärdig. Ersätt *TODO* med din kod!

```
// Returnerar tal mellan 0 och n (exklusive n)
int nyttTal(int n){
   return rand() % n;
}

int anvandarensGissning(){
    // TODO
}

void utforEnSpelomgang(){
   const int n = 100;
   const int datornsTal = nyttTal(n);
   int antalUtfardaGissningar = 0;
   cout << "Datorn tänker på ett tal mellan noll och "<< n << ". Gissa vilket!" << endl;

   // TODO
}
```

```
void ingangTillGissaTal(){
  utforEnSpelomgang();
}
```

3.3 Matteproblemet *3x+1*

I ett inslag på youtube-kanalen *Veritasium* beskrivs nedanstående matematiska serie[1]:

> Börja med ett godtyckligt heltal n. Om talet är udda, multiplicera det med 3 och lägg till 1. Om talet istället är jämnt, dividera det med 2. Upprepa!

Om man exempelvis börjar med n=*7* blir serien:

7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1 (4 2 1 4 2 1 . . .)

Enligt *Collatz Conjecture* kommer serien alltid att nå ner till talet 1 (och sedan fortsätta i den lilla loopen 4 2 1 4 2 1. . .). Vi får anta att Collatz förmodan är sann även om ingen lyckats bevisa att så är fallet.

3.3.1 skrivCollatzSerie

Implementera en funktion

```
void skrivCollatzSerie(int n){
   // TODO
}
```

vars uppgift är att skriva ut talserien fram t.o.m. den första 1:an. Anropa den med en 7:a och verifiera att den skriver ut ovanstående sekvens.

3.3.2 collatzMax

Implementera en funktion

[1] The Simplest Math Problem No One Can Solve, https://www.youtube.com/watch?v=094y1Z2wpJg&t=338s

```
int collatzMax(int n){
   // TODO
}
```

vars uppgift är att hitta det största talet i den serie som börjar med n och returnera detta tal (funktionen skall inte skriva ut något). Om man exempelvis anropar funktionen med en 7:a skall den returnera talet 52 (som ju finns på den sjätte platsen i serien).

3.3.3 collatzLängd

Implementera en funktion

```
int collatzLangd(int n){
   // TODO
}
```

vars uppgift är att ta reda på lång serien är och returnera denna längd. Om man exempelvis anropar funktionen med en 7:a skall den returnera längden 17 (eftersom 1 finns på den 17e platsen).

3.3.4 skrivCollatzStatistik

Implementera en funktion

```
int skrivCollatzStatistik(int maxN){
   // TODO
}
```

vars uppgift är att undersöka alla n-värden mellan 1 och `maxN` och därefter skriva ut det `n`-värde som gav det största max-värdet, och motsvarande max-värde, samt också det `n`-värde som genererade den längsta serien och motsvarande läng.

Testa, vilken utskrift får du när du anropar
`skrivCollatzStatistik(10000)`?

3.4 Matematiska beräkningar

I följande uppgifter skall du programmera några matematiska funktioner. (placera dem i den aktuella cpp-filen enligt anvisningarna i 2.2.9.)

3.4.1 Programmera aritmetisk summa

Enligt den matematiska analysen är

$$1 + 2 + 3 + 4 + \ldots + n \;=\; \frac{n(n+1)}{2} \tag{3.1}$$

Vi skall beräkna summan med en loop och verifiera att vi får ovanstående resultat.

Studera nedanstående kod noggrant, knappa sedan in den och ersätt TODO i `aritmetiskSummaFranLoop` med din egen kod.

```
int aritmetiskSummaFranAnalys(int n){
   return n*(n+1)/2;
}

int aritmetiskSummaFranLoop(int n){
  // TODO
}

void provaSumma(int n){
    double resultat = aritmetiskSummaFranLoop(n);
    double diff = resultat - aritmetiskSummaFranAnalys(n);
    cout << "aritmetiskSummaFranLoop(" << n << ") = " << resultat
         << ", diff = " << diff << endl;
}

void provaFleraSummor(){
    for (int n=0; n<100 ; n+=20)
        provaSumma(n);
}
```

Anropa `provaFleraSummor` och verifiera att analys och loop överensstämmer!

3.4.2 Programmera integral

Enligt den matematiska analysen är

$$\int_a^b x^2 dx = \left[\frac{x^3}{3}\right]_a^b \tag{3.2}$$

$$= \frac{b^3 - a^3}{3} \tag{3.3}$$

Vi skall beräkna integralen med en loop och verifiera att vi får ovanstående resultat.

Studera nedanstående kod noggrant, knappa sedan in den och ersätt TODO i `integralX2FranLoop` med din egen kod (några tips ges efter nedanstående kod).

```
double integralX2FranAnalys(double a, double b){
    return (b*b*b - a*a*a)/3;
}

double integralX2FranLoop(double a, double b){
    // TODO
}

void provaIntegral(double a, double b){
    double resultatLoop = integralX2FranLoop(a,b);
    double resultatAnalys = integralX2FranAnalys(a,b);
    double diff = resultatLoop - resultatAnalys;
    cout << "integralX2FranLoop("<<a<<","<<b<<") = "
         << resultatLoop << ", diff = " << diff << endl;
}

void provaFleraIntegraler(){
    provaIntegral(-1,1);
    provaIntegral(0,1);
    provaIntegral(1,2);
    provaIntegral(2,10);
}
```

Anropa `provaFleraIntegraler` och verifiera att de värden som beräknas av `integralX2FranLoop` verkar stämma (approximativt) med analysen.

Tips. Integralen är detsamma som *arean* under kurvan, vilken vi kan approximera med hjälp av många smala rektangulära staplar.

1. Före nedanstående for-loop:
   (a) Introducera en `const` variabel `dx` för staplarnas bredd (initiera till värdet 0.01).
   (b) Introducera en variabel kallad `area`, med initialvärdet 0.
2. Använd en for-loop av typ: `for (double x=a; x<(b-dx) ; x+=dx)`
3. Inuti for-loopen: Uppdatera `area` med arean av den stapel som har höjden $y = x^2$ och bredden `dx`.

3.4.3 Programmera skrivUtFibonacci

I den talföljd som brukar kallas *fibonacciserien* är de två första talen 1:or, och därefter är varje tal lika med summan av föregående två tal. Här är de 20 första talen i denna sekvens:

> 1:1, 2:1, 3:2, 4:3, 5:5, 6:8, 7:13, 8:21, 9:34, 10:55, 11:89, 12:144,
> 13:233, 14:377, 15:610, 16:987, 17:1597, 18:2584, 19:4181, 20:6765

Implementera nedanstående funktion vars uppgift är att skriva ut de `n` första talen i fibonacciserien enligt ovanstående format (d.v.s. nr:tal, nr:tal, etc.).

```
// Skriver ut de n första fibonacci-talen numrerade från 1.
// skrivUtFibonacci(6) skall skriva ut 1:1, 2:1, 3:2, 4:3, 5:5, 6:8
void skrivUtFibonacci(int n){
    // TODO
}
```

Anropa funktionen och verifiera att den genererar samma sekvens som jag givit ovan.

Tips: En av svårigheterna är att få funktionen att fungera korrekt även om den anropas med ett tal som är mindre än 2. Själv löste jag detta problem med två `if`-satser. Här är de första kodraderna i min egen implementering:

```
if (n>=1)
    cout << "1:1";
if (n>=2)
    cout << ", 2:1";
int x2 = 1;
int x1 = 1;
for (int i=3; i<=n ; i+=1){
    ...
}
```


3.4.4 Programmera kvadratroten

Nedan beskriver jag en iterativ metod för att beräkna $\sqrt{x}$, din uppgift blir sedan att implementera en funktion som gör detta.

Antag att vi vill beräkna kvadratroten ur talet $x = 2$. Vi börjar då med att gissa att kvadratroten är $g = 1$. Därefter beräknar vi kvoten $k = \frac{x}{g}$. Om gissningen är för liten (dvs om $g < \sqrt{x}$) kommer k att bli för stort (dvs $k > \sqrt{x}$) och vice versa. Medelvärdet mellan g och k, $\frac{g+k}{2}$, borde därför bli en bättre gissning.

I ovanstående exempel är $x = 2$ och den första gissningen $g = 1$, vilket leder till kvoten $k = 2$ och den nya bättre gissningen blir därför $\frac{1+2}{2} = 1.5$. Med $g = 1.5$ som gissning blir $k = \frac{2}{1.5} = 1.3333$ och den nya bättre gissningen blir därmed $\frac{g+k}{2} = 1.4166$.

Om vi fortsätter några iterationer bör gissningen konvergera mot $\sqrt{2} \approx 1.4142135$. I din implementering kan du exempelvis iterera 10 varv med en `for`-loop.

Studera nedanstående kod noggrant, knappa sedan in den och ersätt TODO med din egen kod. I `provaKvadratroten` jämförs resultatet från din implementering med standardfunktionen `sqrt`, för att detta funktionsanrop skall fungera måste du ha inkluderat `<cmath>`.

```
// Returnerar kvadratroten ur x (där x>=0)
double kvadratroten(double x){
   // TODO
}

void provaKvadratroten(double x){
    double resultat = kvadratroten(x);
    double diff = resultat - sqrt(x);
    cout << "kvadratroten(" << x << ") beräknad till " << resultat
         << " diff " << diff << endl;
}

void provaFleraKvadratrotter(){
    provaKvadratroten(2);
    provaKvadratroten(10);
    provaKvadratroten(100);
}
```

Anropa `provaFleraKvadratrotter` och verifiera din algoritm fungerar.

3.4.5 Programmera MacLaurin serie

Enligt den matematiska analysen kan alla *snälla* matematiska funktioner skrivas som *MacLaurin* serier[2]. Exempelvis kan e^x skrivas som följande summa:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} \tag{3.4}$$

$$= 1 + \frac{x}{1} + \frac{x \times x}{1 \times 2} + \frac{x \times x \times x}{1 \times 2 \times 3} + \frac{x \times x \times x \times x}{1 \times 2 \times 3 \times 4} + \ldots \tag{3.5}$$

Summan är oändlig, men de första 20 termerna ger förmodligen en ganska bra approximation.

Studera nedanstående kod noggrant, knappa sedan in den och ersätt TODO i `eUpphojtTill` med din egen kod (jag ger några tips efter koden). Funktionens uppgift är att beräkna och returnera e^x med hjälp av de första 20 termerna i ovanstående formel. Koden som följer anropar både din funktion och den "färdiga" funktionen `exp(x)` från mattebiblioteket. (För att anropet till `exp(x)` skall fungera måste man ha inkluderat `<cmath>`.)

```
// Returnerar e upphöjt till x
double eUpphojtTill( double x ){
   // TODO
}

void provaEUpphojtTill( double x ){
    double resultat = eUpphojtTill(x);
    double diff = resultat - exp( x );
    cout << "eUpphojtTill(" << x << ") beräknad till " << resultat
         << " diff " << diff << endl;
}

void provaFleraEUpphojtTill(){
    provaEUpphojtTill(-1);
    provaEUpphojtTill(0);
    provaEUpphojtTill(0.5);
    provaEUpphojtTill(1);
    provaEUpphojtTill(2);
    provaEUpphojtTill(10);
}
```

Provkör `provaFleraEUpphojtTill` och verifiera att felet blir relativt litet (för små x).

[2] $\sum_{n=0}^{\infty} f^{(n)}(0)\frac{x^n}{n!} = f(0) + f'(0)x + \frac{f''(0)}{2!}x^2 + \ldots + \frac{f^{(k)}}{k!}x^k + \ldots$

Tips:

1. Använd en for-loop, `for (int n=1; n<20 ; n+=1){...}`
2. Deklarera följande `double`-variabler utanför loopen:
   (a) `summa`. Initiera med den 0:te termen $\frac{x^0}{0!} = \frac{1}{1} = 1$ och öka den i varje loopvarv. Efter loopen skall summan returneras.
   (b) `nFakultet` Denna variabel skall innehålla talet $n!$ (initiera till 1 och uppdatera i varje loopvarv).
   (c) `xUpphojtTillN` Denna variabel skall innehålla x^n (initiera till 1 och uppdatera i varje loopvarv).

Byt mot `while(true)` och `break`

I ovanstående uppgift fick du anvisningen att iterera 20 varv med en `for`-loop. Om `x` är ett litet tal är det förmodligen onödigt att iterera så många varv, och om `x` är stort är det kanske inte tillräckligt ($20! \approx 2.4 \times 10^{18}$).

Ändra `for`-loopen till en `while(true)` -loop och avbryt loopen med `break` när

$$| \frac{x^n}{n!} | < 0.0001$$

Provkör igen `provaFleraEUpphojtTill`

3.5 Rita med text

Syftet med dessa övningar är att träna på *nästlade loopar*, dvs loopar som innehåller loopar.

Implementera nedanstående funktioner. Ersätt TODO med kod och se till att funktionerna anropas från filens `ingangTill`-funktion.

```
void skrivUtMultiplikationsmatris(int n){
    // TODO
}
void fyllRektangel(int höjd, int bredd){
    // TODO
}

void ritaRektangel(int höjd, int bredd){
     // TODO
}

void fyllTriangel(int höjd){
    // TODO
```

```
}

void fyllCirkel(int radie){
    // TODO
}

void provaRitaFunktionerna(){
    cout << endl;
    ritaMultiplikationsmatris(5);
    cout << endl;
    fyllRektangel(4, 20);
    cout << endl;
    ritaRektangel(4, 20);
    cout << endl;
    fyllTriangel(5);
    cout << endl;
    fyllCirkel(5);
    cout << endl;
}

void ingangTillRitaMedText(){
    provaRitaFunktionerna();
}
```

```
   1   2   3   4   5
   2   4   6   8  10
   3   6   9  12  15
   4   8  12  16  20
   5  10  15  20  25

####################
####################
####################
####################

####################
#                  #
#                  #
####################

#
##
###
####
#####


   #####
  #######
 #########
 #########
 #########
 #########
 #########
  #######
   #####
```

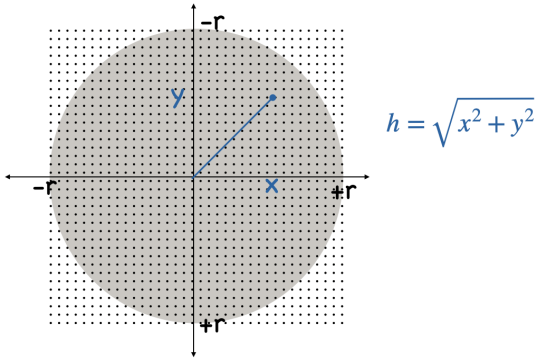
I marginalen ser du vilken utskrift som förväntas.

Tips: Nedan visar jag min egen implementering av `skrivUtMultiplikationsmatris`, vars uppgift är att skriva ut en matris med alla nultiplikationstabeller från 1 till `n` (på rad `r` kolumn `k` skall man kunna hitta produkten `r*k`). Den yttre loopen itererar över rader. För varje rad, `r`, kommer den inre loopen att iterera över kolumner, `k`, och för varje värde på `k` skriver vi ut produkten `r*k` på skärmen med *fix bredd*. Efter det att vi skrivit ut alla produkter på en rad måste vi skapa en radbrytning så att nästa rad hamnar längre ned. Detta gör vi på kodrad 12.

```
7
8  void skrivUtMultiplikationsmatris(int n){
9      for (int r=1; r<=n; r+=1){
10         for (int k=1; k<=n ; k+= 1)
11             cout << setw(4) << r*k;
12         cout << endl;
13     }
14 }
```

3.6 Estimera talet π

En cirkel med radien r har arean $a = \pi r^2$. Vi vet naturligtvis att $\pi \approx 3.1415$, men det kan ändå vara intressant att estimera π från areaformeln.



Ovanstående figur illustrerar hur. Idén är att approximera arean genom att räkna hur många gridkoordinater som ligger innanför cirkeln. Matematiskt liggen en koordinat (x, y) innanför cirkeln om avståndet till origo är mindre än r, d.v.s. om $\sqrt{x^2 + y^2} < r$, d.v.s:

$$x^2 + y^2 \quad < \quad r^2 \tag{3.6}$$

Använd alltså en dubbel for-loop för att loopa igenom alla gridpunkter:

```
for (int y=-r; y<=r ; y+=1)
   for (int x=-r; x<=r ; x+=1){
      ...
   }
```

och räkna hur många av dessa som ligger innanför en cirkel med radien r. Detta ger en uppskattning av arean a. Talet π kan därefter estimeras med a/r^2.

Implementera detta i en funktion `estimatAvPiFrånRadie` (vars uppgift är att beräkna och returnera estimatet av π). Funktionen `provaEstimeraPi` undersöker sedan vilket estimat man får för olika värden på `radie`.

```
double estimatAvPiFrånRadie(int radie){
    // TODO
}

void provaEstimeraPi(){
    for (int radie = 1 ; radie < 10000; radie = radie*2){
        double pi = estimatAvPiFrånRadie(radie);
        cout << " radie = " << radie << ", pi estimeras till " << pi << endl;
    }
}
```

```
void ingangTillEstimeraPi(){
   provaEstimeraPi();
}
```

Förhoppningsvis ser du att estimatet konvergerar mot
$\pi \approx 3.1415926535\ldots$

Litteraturförteckning

[1] Jan Skansholm. C++ direkt, 2011.