TDP004 Objektorienterad programmering

Laborationsmaterial

Kursmaterial till kursen TDP004

Höstterminen 2008

Revision 98

Innehållsförteckning

Introduktion

Detta material innehåller instruktioner för Eclipse samt laborationer och övningar för kursen *TDP004 Objektorienterad programmering*.

Redovisning

Övningar behöver ni **inte** redovisa. Alla uppgifter i en *laboration* **ska** redovisas. Laboration 0 ska visas upp (ni behöver inte skicka in något för denna labb).

Ni laddar upp koden till subversion-arkivet på:

```
https://svn-und.ida.liu.se/courses/TDP004/2008-1-LAB1/grupp_X-Y
```

där X och Y ska ersättas med vilket gruppnummer ni har i webreg.

Efter att ni skickat in koden i subversion ska ni skicka ett e-brev till er laborationsassistent om var i katalogen koden för den aktuella labben finns. **Ämnesraden i det e-brevet ska vara**:

```
TDP004: Labb {LABBNR}
```

Där `{LABBNR}` ersätts med laborationens nummer. Skriv även era namn i e-brevet samt LiU-ID (abcxy123).

Se till så att ni endast skickar in **relevanta filer** i subversionrepositoryt. Det vill säga **inte** genererade filer (*.o, a.out och annat som genereras av komplilatorn och länkaren). Använder du Subclipse enligt instruktionerna nedan så ska det inte vara några problem.

Kodstil

I kursen följer vi Tommy Olssons stilguide, *Enkel stilguide för C++*:

`https://www.ida.liu.se/~tao/pub/lang/cpp/CPP-Enkel-stilguide.pdf`

Se även `www.ida.liu.se/~tao/pub` för mer matnyttigt om C++.

För stora ickemotiverade avsteg från stilguiden ger komplettering.

Utvecklingsmiljö

I kursen används Eclipse som utvecklingsmiljö och Subversion som revisionshanteringssystem. Subversion bör du ha installerat sedan tidigare. Nedan är instruktioner för att installera Eclipse.

Installation

Java

Som standard används libgcj som javamotor (java runtime environment) i Ubuntu. Tyvärr kan det ställa till det för Eclipse och därför vill vi installera Suns JRE (eller alternativt OpenJDK, om man hellre vill det).

Installera paketet 'sun-java6-jre'. För att kontrollera vilken java-version som används kan du köra:

```
$ java -version
```

För att se tillgängliga kör du:

```
$ sudo update-java-alternatives --list
```

För att byta kör du:

```
$ sudo update-java-alternatives -s {JVMNAMN}
```

där du byter ut `{NAMN}` mot till exempel `java6-sun`.

Eclipse

Steg 0

Installera Eclipse och CDT genom Ubuntus pakethanteringssystem (till exempel med *Synaptic*). CDT är en C++-plugin till Eclipse. Paketen heter 'eclipse' respektive 'eclipse-cdt'. Det räcker att installera 'eclipse-cdt' eftersom det ”drar in” paketet 'eclipse' (installerar för närvarande version 3.2.2 av Eclipse).

Detta steg görs mest för att få in de paket Eclipse beror på. Du kan låta bli om du vill men får då själv se till så att behövda paket installeras (se vidare nedan (*)).

Steg 1

Installera Eclipse manuellt (för att få en senare version, version 3.4):

- ladda ner en tarboll från Eclipse hemsida:
  www.eclipse.org --> Downloads --> Eclipse for C++, Linux 64
- packa upp filen och lägg förslagsvis mappen 'eclipse' i '~/opt/'
  (finns inte '~/opt/' så skapa den)
- öppna mappen '~/opt/eclipse' och dra filen 'eclipse' till ditt verktygsfält högst upp på skärmen (gnome toolbar) så kan du enkelt starta Eclipse därifrån

(*) För att kunna kompilera C++-program behöver du en kompilator för ändamålet. Installerar du endast Eclipse manuellt enligt ovan så måste du själv se till så att du har en kompilator. Installera paketet 'g++' så är det fixat! Installera 'gdb' för att kunna debugga program. Se även till så att du har

paketet 'make' installerat samt 'exuberant-ctags' (för indexering).

Steg 2

Starta Eclipse (välj en katalog för workspace – inte viktigt just nu, det går lätt att byta senare) och gå till *Workbench*. Uppdatera:

- Välj *Help --> Software Updates*
- Välj *Update...*
  (vänta på nedladdning)
- Starta uppdateringen (Först *Next*, sen acceptera avtal, sen *Finish*)
- Starta om Eclipse

Uppdatering klar!

Subclipse

Subclipse är en subversion-plugin för Eclipse. För att installera, gör följande (instruktioner för Eclipse version 3.4):

1. Välj *Help --> Software Updates*
2. Välj fliken ”Available Software” och lägg sedan till följande under ”Add Site...”:
   `http://subclipse.tigris.org/update_1.4.x`
3. Nu kan du markera följande för installation (under http://subclipse...):
   – Subclipse (required)
   – SVNkit Adapter (optional)
4. Installera!
   Det kan ta ett tag – kika i statusfältet i Eclipse huvudfönster om du undrar vad som händer.

Nu har du installerat Subclipse och kan börja använda subversion direkt i Eclipse! Först bör du (om du inte redan har gjort det) skapa en katalogstruktur för labbar och övningar i ditt svn-repo, kanske med följande skalkommando (bash):

```
$ svn mkdir --parents URL/labbar/{labb,ovn}{0..5}/uppg{1..8}
```

Byt ut URL mot aktuell URL för repositoryt. Observera att det inte är ett mellanslag direkt efter URL. Du har nu skapat katalogstruktur direkt i repositoryt och kan nu välja lämplig katalog från repot för ditt projekt från Subclipse. Några kataloger är överflödiga eftersom inte alla labbar och övningar har åtta uppgifter – vill du så kan du finslipa lite på kommandot. Till exempel för de två första labbarna och övningarna:

```
$ svn mkdir --parents URL/labbar/labb0
$ svn mkdir --parents URL/labbar/ovn0/uppg{1..6}
$ svn mkdir --parents URL/labbar/ovn1/uppg{1..4}
$ svn mkdir --parents URL/labbar/labb1/uppg{1..5}
```

Användning

För att börja använda Subclipse i ett nytt projekt så gör följande:

1. Välj *File --> New --> Other* (eller tryck Ctrl-N)

2. *SVN --> Checkout Projects from SVN*
3. Lägg till URL för ditt repository (rot-katalogen)
4. Välj katalog från repositoryt att använda för projektet
   (högerklicka för att kunna uppdatera vyn)
5. Använd ”Check out as project configured using the New Project Wizard” (förvalt)
6. Skapa ett C++-projekt:
   C++ --> Executable --> Empty Project

Nu kan du högerklicka på ett projekt och välja ”*Team*” för att hitta subversion-kommandon!

CUTE

Om du vill kan du installera CUTE. För att göra detta, lägg till följande site:

- `http://ifs.hsr.ch/cute/updatesite/`

på samma sätt som för Subclipse. Markera och installera! Se kurshemsidorna för mer info om CUTE (TDP004 och TDP005).

Laboration 0 – Eclipse

Syftet med övningen är att ge en introduktion till Eclipse och CDT. Syftet är också att ni ska känna er bekväma med att börja använda Eclipse som editor samt att ni har tillräckligt med förståelse om Eclipse för att kunna utöka era kunskaper om verktyget på egen hand.

1. Öppna Eclipse och välj en Workspace (platsen där alla era C++-projekt ni skriver i Eclipse hamnar)
2. Första gången ni öppnar Eclipse hamnar ni i ett välkomstperspektiv. För att komma vidare till "standardperspektivet" gå vidare via ”go to the workbench” (motsvarande).
3. Skapa ett nytt C++-projekt. Detta kan du göra på flera sätt, till exempel genom att använda huvudmenyn: *File* --> *New* --> *Project*.
   Välj sedan *C++* --> *Executable* --> *Empty Project*. Du får då upp en dialogruta där du får döpa projektet. Döp det till `test1`. Övriga inställningar behöver inte ändras.
4. Skapa en ny C++-fil genom att högerklicka på ditt projekt och välja *New* --> *Source File*. Döp filen till `hello.cpp`.
   Skriv in koden för ett main-program som skriver ut texten "Hello world!" i filen.
5. Om något är fel i koden visar Eclipse det genom en liten röd ikon i vänsterkanten. Håll musen över ikonen för att se vad felet är. Om du inte fått några fel, prova att införa några, till exempel genom att ta bort semikolon, stava fel, inte stänga paranteser eller "glömma" inkludera paket.
   Man kan även få varningar, en liten gul ikon. Prova till exempel att deklarera en variabel som ni sedan inte använder.
6. Kör programmet.
   Det gör du till exempel genom att välja *Run* --> *Run* (Ctrl-F11) i huvudmenyn.
7. Om du vill köra ditt program igen kan du göra det genom att klicka på den lilla gröna ikonen med en pil, under huvudmenyn.
8. Det mesta i Eclipse går att göra på flera olika sätt. Prova att skapa ett nytt projekt med ett enkelt program, men utför kommandona på något annat sätt. Använd till exempel huvudmenyn om du inte gjort det innan, eller den kontextkänsliga menyn som dyker upp när man högerklickar på olika ställen i Eclipse.
9. Eclipse kan ge dig stöd på olika sätt när du programmerar. Några av dem ska vi titta på nu.
   - Deklarera en sträng i ditt program och ge den ditt namn som värde:
     ```
     string namn;
     namn = "Sara";
     ```
     Ändra i koden så att "Hello [ditt namn]" skrivs ut istället för "Hello world!" som tidigare.
     Högerklicka på variabeln `namn`. Välj *Refactor* i menyn och byt namn på variabeln till `name`.
     Notera att variabelns namn byts ut på alla ställen i koden.
   - Skriv i filen (inklusive punkten):
     ```
     namn.
     ```
     Då får du upp en ruta med förslag på vad som kan göras med en sträng. Titta gärna

igenom alla förslag. Välj sedan `clear`. Skriv ut strängen igen. Vad gjorde `clear`?

- Skriv in bokstäverna `get` i filen. Högerklicka sedan på dem och välj *Source* --> *Content Assist* i menyn (Ctrl-Space). Då visas en lista med alla tillgängliga satser som börjar med "`get`". Titta igenom den, och välj sedan den första förekomsten av `getline`. Lägg till parametrar:
  ```
  getline(cin, name);
  ```
  Skriv ut namn igen. Vad händer när du kör programmet? Jo, det väntar på att du ska skriva in en text. Gör det! Och se vad utskriften blir.
- Prova att använda *Content Assist* på andra bokstavskombinationer och varianter. Skriv till exempel `cout << e` och tryck Ctrl-Space.

10. Det är fortfarande tidigt i kursen och objektorientering har inte kommit upp, men bara för att få en känsla för vad Eclipse kan åstadkomma ska vi prova att skapa en klass. Högerklicka på projektnamnet och välj *New* --> *Class*. Ge klassen ett godtyckligt namn. Titta på filerna som Eclipse skapar för att se vad Eclipse har genererat.
11. Ctrl-Shift-F formaterar enligt vald kodstil (vilket i kursen ska vara K&R – den förvalda). Testa att lägga in radbrytningar lite här och var mellan identifierare och måsvingar och testa sedan kortkommandot (Det blir kanske inte helt perfekt men förhoppningsvis bättre).
12. Ctrl-. (punkt) och Ctrl-, (komma) förflyttar framåt och bakåt mellan bland annat kompileringsfel i en kodfil (röda understrykningar). Kolla i *Edit* (huvudmenyn) för fler kortkommandon och även i kontextmenyn (högerklicka) under *Source*.
13. Tryck F11. Nu får du frågan om du vill byta till ett nytt perspektiv, nämligen perspektivet för debuggning. Svara ja och se hur Eclipse formar om sig till din stora ära! Öhm, jag menar: lägg märke till perspektiv-fliken och perspektiv-knappen uppe i högra hörnet av Eclipse. Där kan du byta mellan olika perspektiv.
    Om du vill kan du nu utforska debugg-perspektivet genom att till exempel stega i programmet med F5 (*step into*) eller F6 (*step over*). Fler möjligheter finns – kolla i fliken Debug uppe till vänster i Eclipse.
    Tryck F8 för att återgå till att köra programmet som vanligt. Byt sedan tillbaka till vanligt perspektiv.
14. Detta var bara ett smakprov på vad som går att göra med Eclipse. Prova gärna att göra fler saker, undersök vilka val som finns i de kontextkänsliga menyerna man får upp när man högerklickar etcetera.
15. När du känner dig klar, visa upp dina projekt för labbassistenten, och var beredd att svara på frågor kring det du gjort i övningen.

Övning 0 – introduktion

Detta är en introduktionsövning som ska göras innan ordinarie laborationer börjar.

Ett mycket viktigt syfte med denna introduktionsövning är att ta upp grundläggande begrepp.

I förklaringarna till exemplen tas en hel del begrepp upp och du förväntas lära dig dessa i samband med denna övning. Du kan behöva läsa i kursboken innan de kommande föreläsningarna om de korta förklaringar som ges här inte gör att du förstår helt.

Denna introduktionsövning syftar till att, som en av de allra första aktiviteterna i kursen, introducera programspråket C++, hur du skriver enkla C++-program och en hel del grundläggande begrepp. Det är också en introduktion till **Eclipse**, **CDT** och **Subclipse**.

På köpet får du även en introduktion till C++-miljön för GNU GCC-kompilatorn g++ – men det primära är att lära dig Eclipse så skippa dessa delar första gången du gör övningen (Emacs/g++).

De olika övningarna omfattar bland annat följande:

- Hur du skriver enkla program i C++.
- Hur du översätter källkod till objektkod och körbar kod med kompilatorn g++ och i Eclipse.
- Hur du kör program.
- Hur du använder standardbiblioteket för C++, bl.a. in- och utmatning med strömbiblioteket.
- Exempel på uttryck och operatorer.
- Exempel på satser (**if**, **for**).
- Exempel på funktioner och funktionsparametrar.
- Hur du hittar information.

Målet är att försöka hinna med dessa uppgifter under den första veckan innan ordinarie laborationer sätter igång.

Uppgift 1 – Ett enkelt program

Eclipse

Starta Eclipse. Skriv följande program och spara det på en fil med namnet `uppg1.cpp`. Har du följt eclipse-instruktionerna så har du redan en katalogstruktur för dina labbar och övningar annars bör du fixa det nu.

Följande kommando skapar kataloger under *ovn0* för respektive uppgift som du kommer skriva i denna övning (som brukligt får du också skriva en kommentar):

```
$ svn mkdir --parents URL/labbar/ovn0/uppg{1..6}
```

Finns inte `ovn0` så skapas den (`svn help mkdir` ;-))

Skapa projektet:

- *File --> New --> Other...*
- *SVN --> Checkout Projects from SVN*

- Välj repository (lägg till om nödvändigt)
- Välj katalogen `labbar/labb0/uppg1`
- Gå vidare med det förvalda ”*Check out as a project configured using the New Project Wizard*”
- Välj *C++* --> *C++ Project*
- Skriv ett projektnamn, till exempel *uppg1*, och välj sedan *Executable* --> *Empty Project*
- Gå vidare med *Finish*

Du har nu ett tomt projekt som du kan skapa nya filer i. Skapa filen `uppg1.cpp` genom att:

- Välja *File* --> *New* --> ***Source Folder*** och skapa katalogen *src*.
  Detta steg är inte nödvändigt, speciellt inte i introduktionsövningarna (vi skapar ju bara en kodfile per projekt), men trevligt annars då vi kommer ha flera kodfiler. Du kan ju testa en gång för att se hur det funkar men skippa det för övriga ”programX”-projekt.
- Välja *File* --> *New* --> ***Source File*** skapa filen `uppg1.cpp`. Om du har en `src`-katalog se till så att filen hamnar däri.

Emacs

Starta Emacs, skriv följande program och spara det på en fil med namnet `uppg1.cpp`. Det bästa sättet att, i detta fall, starta Emacs är att även ange filnamnet på kommandoraden. Då kommer Emacs även att hamna i rätt redigeringsmod, den för C++.

uppg1.cpp

```
/*
 * Program som skriver ett meddelande på standard utmatningsfil.
 */
#include <iostream>
using namespace std;

int main()
{
     cout << "C++ is so cool!" << endl;
     return 0;
}
```

Eclipse

För att kompilera och länka kan du till exempel välja *Project* --> *Build Project*. För att köra kan du trycka på den gröna ”play”-knappen i verktygsfältet eller välja *Run* --> *Run* (Ctrl+F11).

Subclipse

Högerklicka på projektet i *Project Explorer* och välj *Team* --> *Commit* och gå vidare där för att lägga in projektet i subversionrepositoryt. Skriv en kommentar och kolla vilka filer som ändrats i och som kommer läggas till.

I *Project Explorer* kan du nu se revisionsnummer, datum, klockslag och vem som ändrat i filen till

höger om namnet (inom hakparenteser och i en lite gulare färg). För projektet visas vilken katalog den ligger i relativt roten för repositoryt.

g++

Kompilera källkodsfilen med skalkommandot:

```
$ g++ uppg1.cpp
```

Kör programmet med skalkommandot:

```
$ ./a.out
```

Lärdom

Vad kan man lära sig av detta, förutom hur man kompilerar och kör ett vanligt C++-program?

- ett program består alltid av en eller flera **funktioner**. I detta fall en enda funktion som heter main.
- ett program ska alltid ha en funktion som heter `main`. Det är i `main` som exekveringen, körningen, av programmet startar. Du kan se det som om det bakomliggande exekveringssystemet för att köra program anropar funktionen `main`.
- `main` ska alltid returnera ett heltalsvärde; värdet 0 betyder att allt har gått väl. Det är fel att, som i boken *C++ direkt*, deklarera `main` utan returtypen; en returtyp ska alltid anges och vara `int` för `main`.
- `iostream` är en **inkludering** av en del av **standardbiblioteket** som kallas **strömbiblioteket**, och som innehåller funktionalitet för att läsa och skriva s.k. **strömmar**.
- Standardströmmen för utskrifter heter `cout`. När man skriver text i utströmmen `cout` med utskriftsoperatorn <<, skrivs texten ut i det terminalfönster där programmet körs.
- `endl` tillhör också strömbiblioteket och är en **manipulator**, som har den "manipulativa effekten" att sätta in ett nyradstecken i strömmen `cout`.
- `/*...*/` är en form av **kommentar**, där `/*` inleder kommentaren och `*/` avslutar; all text däremellan utgör kommentar, och sådana är endast är till för den mänskliga läsaren av programkoden.

Alla namn i standardbiblioteket är kapslade **namnrymden** `std`. En namnrymd är ett slags modul som innesluter, kapslar, de deklarationer som ingår i namnrymden så att de inte är direkt synliga. Ett sätt att göra namnen i en namnrymd synliga i ett program är att öppna namnrymden med direktivet `using namespace`.

Eclipse

För att städa i projektet välj *Project --> Clean...* När man städar ett projekt tas de filer bort som genereras när man bygger projektet (till exempel *.o).

g++

Städa filkatalogen och kompilera om programmet med kommandona:

```
$ clean
$ g++ -std=c++98 -pedantic -Wall -Wextra -o uppg1 uppg1.cpp
```

Det körbara programmet erhålls nu på en fil med namnet `uppg1`. Kör programmet!

Frågor och uppgifter

- Ändra i programmet så det skrivs ut en tom rad både före och efter meddelandet som skrivs ut.

Uppgift 2 – Variabler och deklarationer

Eclipse

Skapa ett nytt projekt med namnet `uppg2` och skapa kodfilen `uppg2.cpp`.

Emacs

Kopiera filen `uppg1.cpp` till en ny fil med namnet `uppg2.cpp` och ändra enligt följande.

uppg2.cpp

```
// Skriver ett meddelande på standard utmatningsfil.
#include <iostream>
#include <string>
using namespace std;

int main()
{
     string  message = "C++ is so cool!";
     cout << endl << message << endl << endl;
     return 0;
}
```

Kompilera programmet så att det körbara programmet får namnet message och provkör.

Vad kan man få ut av detta program?

- Vi har ett exempel på den andra formen av kommentar som finns, en **radslutskommentar**. En sådan inleds med `//` och avslutas då raden tar slut.
- Inkluderingen av `string` görs därför att en **datatyp** som heter `string` ska användas i programmet. En datatyp anger en viss *typ* av värden, t.ex. heltal eller strängvärden, och vilka **operationer** som är tillåtna på sådana värden, t.ex. aritmetiska operationer för heltal.
- Även `string` tillhör standardbiblioteket och blir synlig genom direktivet **using namespace**.
- `string` är en datatyp för **variabler** (och konstanter) som ska innehålla textsträngar, dvs en följd av tecken av typen `char`. En variabel är en behållare för ett värde av en viss datatyp.
- `"C++ is so cool!"` är ett uttryckligt strängvärde. Ett sådant uttryckligt värde (inte

bara av strängtyp) kallas **litteral**.

- Alla variabler i ett program måste införas genom en **deklaration**. Variabeln `message` deklareras i detta fall som en **lokal variabel** i funktionen `main`. Det betyder att den har sin **räckvidd** inom `main` och inte är (och aldrig kan bli) synlig utanför `main`.
- En variabeldeklaration, som den för `message`, är en deklaration som genererar variabelns minnesutrymme. En sådan deklaration är också en **definition**. (I lite större program behöver man ibland deklarera att en variabel med ett viss namn finns definierad på ett annat ställe i programmet och en sådan deklaration är då inte en definition.)
- I detta fall **initieras** `message` (ges ett startvärde) direkt i samband med definitionen, vilket är något man ska eftersträva att göra. Det i detta fall, sämre alternativet vore att först deklarera variabeln och sedan tilldela den ett värde i en efterföljande **tilldelning**, vilket görs med **operatorn** =.

```
string  message;
message = "C++ is so cool!";
```

- Variabler kan initieras automatiskt. Det finns regler för i vilka sammanhang variabler initieras automatiskt och i så fall med vilket värde (i princip ett 0-värde för datatypen ifråga).
- En datatyp som string är en avancerad typ och dess variabler initieras alltid automatiskt till tomma strängen, dvs en sträng som inte innehåller något tecken och som alltså motsvarar **stränglitteralen** "".

Frågor och uppgifter

- Ändra i programmet så att utskriften blir:

```
Message of the day: C++ is so cool!
```

Uppgift 3 – Inmatning via tangentbordet

Eclipse

Skapa nytt projekt och fil som tidigare.

Emacs

Städa i filkatalogen, kopiera sedan föregående program till en fil med namnet `uppg3.cpp` och ändra enligt nedan.

uppg3.cpp

```
#include <iostream>
#include <string>
using namespace std;

const string  message = "C++ is so cool!";

int main()
{
     string  first_name;

     cout << "Skriv ditt förnamn: ";
     cin >> first_name;

     cout << first_name << " says: " << message << endl;

     return 0;
}
```

Kompilera och provkör programmet! Prova olika sätt att mata in ditt förnamn, t.ex. genom att skriva mellanrumstecken och även slå RETURN-tangenten innan du skriver ditt namn.

Vad är nytt i detta program?

- Vi har gjort `message` till en **konstant** sträng genom att deklarera **const** före datatypen `string`.
- En konstant måste alltid initieras i definitionen och kan sedan inte ändras. `message` är ett *namngivet* konstant värde, en **symbolisk konstant**, till skillnad från `"C++ is so cool!"` som är en **litteral konstant**.
- Konstanten `message` har också flyttat ut ur funktionen `main` och **deklarerats på filnivå**. Genom detta är dess räckvidd potentiellt hela programmet. Det betyder att om det skulle finnas fler funktioner, och även om dessa funktioner skulle finnas på flera filer, skulle `message` kunna kommas åt från samtliga dessa funktioner.
- Vi har deklarerat en ny variabel `first_name`, för att lagra ett förnamn.
- `cin` är standardströmmen för inmatning och den är kopplad till tangentbordet.
- för att läsa data från `cin` använder vi **inläsningsoperatorn** >> i ett uttryck där vi läser in förnamnet och får det lagrat i variabeln `first_name`.
- Operatorn >>, liksom operatorn <<, gör s.k. **formaterad utskrift**, respektive **formaterad inläsning**. I detta fall, då vi läser strängvärden, innebär det:
  1. först läses eventuella inledande **vita tecken** (mellanrum, tabulertecken, radslut) som skrivits i inströmmen förbi,
  2. sedan läses tecken för tecken och lagras i variabeln, till dess det kommer ett vitt tecken eller det blir slut i inströmmen.
- Vi har delat upp koden i `main` i fyra avsnitt med hjälp av tomma rader: variabeldeklaration, inläsning av förnamnet, utskrift av meddelandet, samt avslutning. Detta ökar läsbarheten,

genom att vi enkelt kan urskilja de fyra delmomenten.

Frågor och uppgifter

- Ändra i programmet så att du kan mata in både ditt förnamn och efternamn i var sin variabel. Låt programmet skriva ut dessa snyggt i meddelandet.
- Prova lite olika sätt att skriva in ditt förnamn och efternamn, t.ex. på samma rad med ett eller flera mellanrumstecken före, mellan och efter namnen; på olika rader, etc., för att testa den formaterade inmatningen.

Uppgift 4 – Funktioner och funktionsparametrar

Eclipse

Skapa projekt och källkodsfil som tidigare. Provkör.

Emacs

Skriv följande program i en fil med namnet `uppg4.cpp`. Kompilera och provkör.

uppg4.cpp

```
#include <iostream>
#include <iomanip>
using namespace std;

void print_squares(int n)
{
     cout << endl << setw(6) << "i" << setw(12) << "i * i"
          << endl << endl;

     for (int i = 1; i <= n; ++i)
          cout << setw(6) << i << setw(12) << i * i << endl;
}

int main()
{
     int  x;

     cout << "Ge ett positivt heltal: " << endl;
     cin >> x;

     if (x < 1)
          cout << "Det var inget positivt heltal!" << endl;
     else
          print_squares(x);

     return 0;
}
```

Vad är nytt?

- Vi har ett program som består av två funktioner, `main` och `print_squares`, som **anropas** i `main`.
- Returtypen `void` innebär att funktionen `print_squares` inte returnerar något värde alls.
- **Parametern** `n` till `print_squares` har typen **int**, en **heltalstyp** som tillhör de **enkla datatyperna**.
- Manipulatorn `setw` (*fältvidd*) bestämmer **fältvidden** för den efterföljande utskriften. Värdet skrivs normalt ut högerjusterade i fältet.
- Vi måste inkludera `iomanip` då vi använder **parametriserade manipulatorer**, som `setw`, däremot inte för oparametriserade manipulatorer, som `endl`.
- **for**-satsen är ett exempel på en **repetitionssats**. Den används ofta för att stega igenom en följd av värden, som i detta fall stega styrvariabeln `i` från `1` till värdet i parametern `n`.
- **if-else**-satsen är ett exempel på en **selektionssats**. Sådana satser används för att styra vilken väg programexekveringen ska ta, i detta fall om en felutskrift ska göras eller om `print_squares` ska anropas.

- Det uttryck som står mellan parenteserna efter **if** är **styruttrycket**: om det blir *sant* utförs satsen efter **if**, om det blir *falskt* utförs satsen efter **else**.
- I detta fall är styruttrycket ett **relationsuttryck** med operatorn < (mindre än). Vilka slags uttryck som kan användas som styruttryck är en historia för sig, liksom vad som kan tolkas som sant och falskt...
- I **if**-satsens **else**-gren finns anropet av `print_squares`, där variabeln `x` ges som argument till parametern `n`. I detta fall kopieras `x`:s värde till parametern `n` i samband med anropet.

Frågor och uppgifter

- Provkör med ett *stort* heltalsvärde och notera vad som skrivs ut. Kan du förklara resultatet?
- Ändra i `print_squares`, så att utskrift endast görs om `n` är mindre än eller lika med något bra värde...

Uppgift 5 – Kommandoradsargument

Eclipse

Skapa projekt och källkodsfil som tidigare.

Emacs

Skriv följande program och spara på filen `uppg5.cpp`.

uppg5.cpp

```
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
     if (argc == 1)
          cout << "C++ is so cool!" << endl;
     else
          for (int i = 1; i < argc; ++i)
               cout << argv[i] << endl;
}
```

Vad är nytt?

- `main` har i detta fall försetts med två parametrar, `argc` och `argv`. Genom dessa överförs värden till `main` från exekveringssystemet. Det är definierat i C++ att `main` kan ha just dessa två parametrar (och även en tredje vid behov för att komma åt miljövariabler, samt inga alls, som vi sett tidigare).
- Via parametern `argc` ("*argument count*") och `argv` ("*argument vector*") kan programmet komma åt text som skrivits på kommandoraden, inklusive själva programnamnet.
- `argc` anger hur många "ord" som skrivits på kommandoraden.

- Varje "ord" finns som en separat C-sträng (se nedan) i fältet `argv`.
- Ett **fält** (eng. "array") är en sammansatt datatyp där elementen är av samma typ, i detta fall **char**.
- Endimensionella fältvariabler definieras på formen:
  elementdatatyp fältnamn[dimension]
- När man deklarerar ett fält, som den formella parametern `argv`, behöver man inte ange någon **dimension** och det skulle dessutom inte användas på något sätt.
- De enskilda elementen i ett fält åtkoms via **indexering** med **indexeringsoperatorn** `[]`.
- Första index i fält är alltid `0` (det förklarar styruttrycket "i < `argc`" i `for`-satsen; de `argc` stycken "orden" på kommandoraden finns på index `0` till `argc`-1 i fältet `argv`).
- **char*** är en strängdatatyp som finns i C och som också var den strängtyp som från början fanns i C++.
- **char*** betyder egentligen typmässigt "pekare till tecken" men normalt pekar en sådan pekare (adress) på det första tecknet i en textsträng av typen **char**[], "fält av tecken"; en s.k. C-sträng.
- **char*** och **char**[] är lågnivåsträngtyper och en ökänd källa till fel i C- och C++-program. Dessa har i C++ kompletterats (läs gärna ersatts) med den fullfjädrade strängtypen `string`.

Frågor och uppgifter

- Vad innebär det att `argc` är lika med `1` när man kör ett program?
- Ändra i programmet, så att alla kommandoradsargument skrivs ut på en enda rad, med ett mellanrumstecken mellan varje argument.
- Ändra i programmet, så att alla kommandoradsargument skrivs ut på en enda rad, men i omvänd ordning.

Eclipse

För att ange argument till programmet gör följande:

- Välj *Run --> Run Configurations*
- Hitta *uppg5* och fliken *Arguments*
- Skriv in texten i textfältet!

Om du vill kan du testa att duplicera en befintlig konfiguration och sedan ändra argumentet för att snabbt ha två olika uppsättningar argument att laborera med.

Emacs

Kompilera programmet och provkör det på två sätt:

```
$ ./uppg5
$ ./uppg5 C++ är så koolt!
```

Uppgift 6 – En liten klass

Skapa projektet uppg6 och en källkodskatalog `src`. Vidare skapar du en källkodsfil med namnet `uppg6.cpp` vari du gör en `main`-funktion enligt nedan.

Skapa även en ny klass genom att välja *File --> New --> Class* och namnge den `Hello`. Nu har du fått två till filer i din src-katalog (om du såg till så att klassens filer hamnade däri) `Hello.cpp` och `Hello.h`. Om du står i en av dessa filer och trycker *Ctrl-Tab* kommer du byta till den andra. *Alt-Pil vänster* och *Alt-Pil höger* flyttar dig bakåt och framåt bland de filer du editerar. *Ctrl-Page Down* respektive *Ctrl-Page Up* flyttar dig höger och vänster bland editor-flikarna (precis som i till exempel Firefox).

Ändra i klass-filerna enligt nedan.

Skapa en publik metod i `Hello` som heter `say_hello` och som skriver ut strängen ”`Hello!`” på standard utmatningsfil med avslutande radbrytning.

Skapa i `main` en instans av `Hello` och anropa sedan `say_hello` för den instansen. Se till att inkludera behövda filer!

uppg6.cpp

```
#include "Hello.h"

int main()
{

     Hello lo;
     lo.say_hello();

     return 0;
}
```

Hello.h

Lägg till **deklarationen** av metoden `say_hello` (efter deklarationen av destruktorn `~Hello`):

```
     void say_hello();
```

Hello.cpp

Lägg till följande inkludering (ovanför inkluderingen av ”Hello.h”):

```
#include <iostream>
```

Lägg till **definitionen** av metoden `say_hello`:

```
void Hello::say_hello() {
     cout << "Hello!" << endl;
}
```

Övning 1

Uppgift 1 – Variabler

Uppgift 1.1

Gör tre program som lagrar heltalet 42 i en variabel och sedan skriver ut det. Initieringen av variabeln ska ske enligt följande tre olika mönster:

a) *typ variabel* = *värde*;

b) *typ variabel*(*värde*);

c) *typ variabel*;
variabel = *värde*;

Uppgift 1.2

Gör a) och b) från föregående uppgift, men deklarera denna gång variabeln som konstant (const).

Att fundera på: Varför fungerar inte fall c) med konstant variabel?

Uppgift 2 – Aritmetik, uttryck och styrstrukturer

Uppgift 2.1

Undersök vad variabeln y innehåller efter att följande bit kod har körts:

Att fundera på: Kommer y innehålla 9 eller 7? Varför blir det som det blir?

```
int x, y;
x = 4;
y = 5 + x;
x = 2;
```

Uppgift 2.2

Gör ett program som beräknar och skriver ut följande uttryck:

startvärden: `int x = 3, y = -7, z = 13;`

a) -x
b) z – x
c) x + y + z
d) x * y + z
e) z + x * y
f) x * (y + z)
g) z / x
h) z % x

Uppgift 2.3

Gör ett program som beräknar följande uttryck och skriver ut variablernas innehåll efteråt:

a)
```
int x = 5;
x += 2;
```

b)
```
int x = 100;
x -= 12;
```

c)
```
int x = 4;
x *= 10;
```

d)
```
int x = 60;
x /= 3;
```

e)
```
int x = 17;
x %= 12; // Tänk klocka...
```

Uppgift 2.4

Gör ett program som beräknar följande uttryck och skriver ut variablernas innehåll efteråt:

a)
```
int x, y;
x = 10;
y = x++;
```

b)
```
int x, y;
x = 10;
y = ++x;
```

c)
```
int x, y;
x = 10;
y = x--;
```

d)
```
int x, y;
x = 10;
y = --x;
```

Uppgift 2.5

Gör ett program som beräknar och skriver ut följande uttryck:

bibliotek:
```
#include <cmath>
```

startvärden:
```
double x = M_PI / 6.0;
double y = M_PI / 3.0;
```

a) sin(x)

b) sin(y)

c) cos(x)

d) cos(y)

Uppgift 2.6

Undersök i vilken ordning variablerna skrivs ut i följande sats.

Bibliotek:
```
#include <iostream>
using namespace std;
```

```
int x = 17, y = 42;
cout << x << endl << y << endl;
```

Uppgift 2.7

Undersök i vilken ordning variablerna läses in i följande sats.

Bibliotek:
```
#include <iostream>
using namespace std;
```

```
int x, y, z;
cin >> x >> y >> z;
```

Uppgift 2.8

Använd if-satser för att undersöka vilka av följande villkor som är uppfyllda.

Startvärden: `int x = 3, y = 4, z = -3;`

a) (x == y)
b) (x != z)
c) (x > y)
d) (x >= x)
e) (z < y)
f) (y <= x)

Uppgift 2.9

Använd **if**-satser för att undersöka vilka av följande villkor som är uppfyllda.

Att fundera på: Vad betyder likhet respektive olikhet mellan två sanningsvärden?

Startvärden: `int x = 3, y = 4, z = -3;`

a) ((y < z) && (y > x))
b) ((x == z) || (x == -z))
c) ((x > y) == (x < z))

Uppgift 2.10

Gör ett enkelt program som först läser in ett heltal och sedan skriver ut ”jämnt” om talet är jämnt respektive ”udda” om talet är udda.

Tips: Modulooperatorn kan användas till att kolla om ett heltal är jämnt, eftersom resten vid heltalsdivision med 2 anger om talet är jämnt eller udda.

Uppgift 2.11

Använd `for`-loopar för att göra ett program som skriver ut alla tal från och med 1 till och med 10.

Uppgift 2.12

Använd `for`-loopar för att göra ett program som läser in ett heltal, räknar ut fakulteten av det och skriver ut det. Fakulteten av ett tal `n` beräknas genom att multiplicera alla tal från och med 1 till och med n. Kontrollera ditt program med följande in- och utdata:

In	*Ut*
0	1
1	1
2	2
3	6
4	24

5	120
6	780

Uppgift 2.13

Lös uppgift 2.11, men använd denna gång `while`-loopar.

Uppgift 2.14

Lös uppgift 2.11, men använd denna gång `do while`-loopar.

Uppgift 2.15

Använd switch-satser för att göra ett program som läser ett heltal och ger följande utdata:

- om talet är 1: skriv ut ”norr”
- om talet är 2: skriv ut ”öster”
- om talet är 3: skriv ut ”söder”
- om talet är 4: skriv ut ”väster”
- övriga fall: skriv ut ”fel”

Uppgift 3 – Funktioner

Uppgift 3.1

Skriv en funktion med namnet *haelsning* som varken tar några inparametrar eller har något returvärde. Det enda funktionen skall göra är att skriva ut texten ”Hej, din gamle galosch!”. Skriv sedan ett program som anropar funktionen.

Uppgift 3.2

Skriv en funktion med namnet `flera_haelsningar` som tar ett heltalsargument `n`. Funktionen ska anropa `haelsning` från föregående uppgift n antal gånger. Även denna funktion skall inte returnera något. Gör till sist ett program som läser in ett tal och anropar `flera_haelsningar` med detta.

Uppgift 3.3

Studera följande funktion:

```
void nollstaell(int x)
{
      x = 0;
}
```

Denna funktion fungerar dock inte som den ska. Vad är fel och vad behöver ändras för att den ska fungera?

Uppgift 3.4

Skriv en funktion `inc` som tar ett heltalsargument `var` och som inte returnerar något. Funktionen skall stega upp variabeln som ges som argument med ett. Använd följande program för att testa om funktionen fungerar ordentligt:

```
int main()
{
     int x = 5;
     int foere, efter;

     foere = x;
     inc(x);
     efter = x;

     if (efter == foere + 1)
          cout << "inc fungerar" << endl;
     else
          cout << "inc fungerar ej" << endl;

     return 0;
}
```

Uppgift 3.5

Skriv en funktion `kvadrat` som tar ett heltalsargument `x` och returnerar kvadraten av `x`.

Uppgift 3.6

Skriv en funktion `kelvin` som tar ett flyttalssargument `c` (grader celcius) och returnerar temperaturen i kelvin (som ett flyt-tal).

Notering: 0 K = -273,15 °C

Uppgift 3.7

Skriv en funktion `haelsing_delux` som tar ett `bool`-argument `ny_rad` med standardvärdet `true`. Funktionen ska fungera precis som funktionen `haelsning` ovan, förutom att man ska kunna välja om ett nyradstecken ska skrivas ut eller ej.

Uppgift 3.8

Skriv en funktion `personlig_haelsning` som tar ett argument `namn` av typen `string`. Funktionen ska skriva ut en text enligt formatet ”Hej *namn*, din gamle galosch!”. Argumentet `namn` får inte kopieras när funktionen anropas och får inte ändras under funktionens körning. Gör också ett program som frågar efter användarens namn och skickar med det till funktionen.

Uppgift 4 – Klasser

(C++ Primer: kap 1.5, sid 20; kap 2.8, sid 63; Part III)

Uppgift 4.1

Skriv en klass:

- Klassen `Animal`
  1. Attribut
     `private string _name;`
     - Sparar namnet.
     - Initieras när man skapar en instans av klassen (Primer sid 263).
  2. Metod
     `protected string get_type();`
     - Returnerar djurets typ som en sträng – här ”Animal”.
  3. Metod
     `public void say_hello()`
     - Skriver ut ”<TYP> NAMN: says hello!” där TYP hämtas från `get_type` och NAMN från `_name`.

Exempel:

```
int main() {

    Animal fido("Fido");
    Animal razer("Razer");

    fido.say_hello();
    razer.say_hello();

}
```

Skriver ut:

```
<Animal> Fido: says hello!
<Animal> Razer: says hello!
```

Laboration 1

Uppgift 1 – Iterering över tal

Skriv ett program som utför följande:

1. Läs in ett start-tal och ett slut-tal.
2. Kontrollera att slut-talet är större än start-talet och att båda talen är större än noll. Om så inte är fallet, skriv ut ”Inmatningsfel!” och avbryt programmet.
3. Iterera igenom alla heltal mellan start-talet till och med slut-talet och skriv ut de tal som uppfyller följande krav:
   - Talet är jämnt delbart med 2.
   - Talet är jämnt delbart med 3 eller 5.

En körning av programmet ska kunna se ut som i följande **exempel**:

```
Ange start: 50
Ange slut: 100
50 54 60 66 70 72 78 80 84 90 96 100
```

Uppgift 2 – Miniräknare

Den här gången ska du programmera en mycket enkel miniräknare. Den ska fungera genom att man har en variabel som lagrar ett heltal (som vid programstart är satt till noll). Med detta tal kan man sedan från en textbaserad meny välja att utföra en av fem operationer: addition, subtraktion, multiplikation, division eller avsluta. Operationen utförs mellan det talet som ligger lagrat i variabeln och ett tal som användaren matar in efter instruktionen. Resultatet lagras därefter i variabeln. Efter att instruktionen har utförts visas innehållet i variabeln och programmet återgår tillbaka till val av operation. Programmet fortlöper tills användaren väljer att avsluta det.

Körexempel som demonstrerar hur programmet ska kunna användas:

```
Operationer:
1. Addition
2. Subtraktion
3. Multiplikation
4. Division
5. Avsluta
= 0
Välj operation: 1
Addera med: 10
= 10
Välj operation: 3
Multiplicera med: 3
= 30
Välj operation: 4
Dividera med: 5
= 6
Välj operation: 7
Felaktigt val!
= 6
Välj operation: 5
Avslutar...
```

Menyn i början skall skrivas ut från programmet. Implementera programmet med `do while`-loopar och `switch`-satser.

Uppgift 3 – Perfekta tal

Uppgiften är att skriva ett program som räknar ut alla perfekta tal upp till en övre gräns. Ett perfekt tal är ett tal där summan av alla dess delare är talet själv. Ett exempel är talet 6, som delas av 1, 2 och 3 (talet 6 själv räknas inte). Summerar man de talen så får man 6, i detta fallet samma som talet själv, vilket betyder att talet 6 är ett perfekt tal.

När programmet startas ska användaren få ange en övre gräns. Programmet ska sedan iterera från 1 till den övre gränsen och skriva vilka av de talen som är perfekta. En enkel algoritm för att undersöka om talet *n* är perfekt är denna: Iterera igenom alla tal från 1 till *n* - 1. Är talet jämnt delbart med *n* så lägg till det till en summavariabel. Är summan sedan lika med *n* så är *n* ett perfekt tal.

Programmet ska kunna användas på följande sätt: (Användarens indata markerat med fet stil)

```
Ange övre gräns: 1000
Perfekta tal: 6 28 496
```

Uppgift 4 – Sparande på bankkonto

Du ska göra ett program som utifrån ett av användaren inmatat startkapital, önskat kapital och en inmatad årsränta kan räkna ut hur många hela år det tar att få det önskade kapitalet. Problemet skall lösas genom att dela upp huvudprogrammet i funktioner (t.ex. för inmatning, indatakontroll och beräkning). Årsräntan anges i procent. (Tips: omvandla räntan till en förändringsfaktor.)

Körexempel: (Användarens indata markerat med fet stil)

```
Ange startkapital: 1000.00
Ange önskat kapital: 1100.00
Ange årsränta: 2.1
Det tar följande antal år att uppnå det önskade kapitalet: 5
```

Programmets indata ska kontrolleras efter följande krav:

- Inget angivet kapital får vara negativt.
- Det önskade kapitalet måste vara större än eller lika med startkapitalet.
- Årsräntan måste vara större än noll.

Om något av kraven inte är uppfyllt ska texten ”Inmatningsfel!” skrivas ut och programmet sedan avslutas, efter att användarens inmatningar är klara. Ett förtydligande exempel:

```
Ange startkapital: 8000.00
Ange önskat kapital: 3000.00
Ange årsränta: 5.0
Inmatningsfel!
```

Uppgift 5 – Robot

Skriv följande klass:

- Klassen `Robot`
  1. Attribut
     `private string name;`
     - Sparar namnet på roboten.
  2. Attribut
     `bool running;`
     - Ska vara falskt som default.
  3. Konstruktor:
     Ska anropas med `string name` som argument vilket anger robotens namn
     - Sparar angivet namn i `name`.
  4. Metod
     `void start();`
     - Om `running` är falskt så ska följande ske:
       1. `running` sättas till sant.
       2. Skriver ut ”Starting <name>.” där <name> är namnet man angav som argument till konstruktorn.
     - Om `running` redan är sant så ska antingen texten ”<name>: \”Stop poking me!\”” eller en av två andra (lämpliga) texter du hittar på skrivas ut – slumpa fram vilken det blir. Använd konstanter.
  5. Metod
     `void stop();`
     - Om `running` är sant så ska följande ske:

1. `running` sättas till falskt.
2. Skriver ut ”<name> stopped.” där <name> är namnet man angav som argument till konstruktorn.

- Om `running` redan är falskt så ska texten ”<name>: \”Zzzz...\”” skrivas ut.

6. Metod
   ```
   bool is_running();
   ```
   - returnerar sant om man startat roboten annars falskt

Gör ett program som slumpmässigt startar och stoppar några robotar mellan 10 och 20 gånger vardera.

Övning 2 – STL

Uppgift 1 – String

Uppgift 1.1

Skriv ett program som skapar en sträng A med ”*Här finns inga*”, och en sträng B med ”*flitiga studenter*”. Ta bort ordet *inga* ur A, sätt ihop sträng A med B samt skriv ut resultatet.

Uppgift 1.2

Skriv ett enkelt krypteringsprogram som läser in en mening i en sträng och skiftar alla tecken med ett steg åt höger. Exempelvis blir ”*hemligt*” krypterat ”*ifnmjhu*”.

Uppgift 1.3

Skriv ett program som läser in ett ord och kontrollerar om huruvida är ordet är ett palindrom (t.ex. *latmasksamtal* är ett palindrom medan *bäversvans* inte är det).

Uppgift 2 – Vector

Uppgift 2.1

Lägg till ett heltal till en vektor. Skriv ut vektorns storlek och kapacitet. Lägg till två heltal till och skriv återigen ut vektorns storlek och kapacitet. Notera hur storleken och kapaciteten ändras.

Uppgift 2.2

Läs in fem heltal till en `vector` genom att använda `push_back`. Ta bort elementet längst fram och längst bak. Stega sedan igenom vektorn med hjälp av en iterator och skriv ut alla tal.

Uppgift 2.3

Gör om ovanstående uppgift men använd istället en `vector` med pekare till heltal (`int`), dvs. `vector<int*>`.

Uppgift 3 – List

Uppgift 3.1

Skriv ett program som skapar en lista med heltalen 1, 2, 3 och en lista med 2, 3, 4. Programmet skall sedan använda lists egna funktioner för att sammanfoga båda listorna samt ta bort alla dubbletter. Skriv ut elementen i den resulterande listan.

Uppgift 3.2

Skriv ett program som skapar en lista med 5 stycken heltal. Använd sedan `find_if` för att undersöka om sekvensen innehåller några jämna tal.

Tips! Använd `%`-operatorn.

Uppgift 4 – Sort

Uppgift 4.1

Läs in talen 1, 5, 2, 4, 3 i en `vector` och sortera dessa genom att använda sort. Skriv sedan ut elementen i den sorterade vektorn.

Uppgift 4.2

Gör om ovanstående uppgift men använd istället en lista och dess motsvarande `sort`-funktion

Uppgift 4.3

Läs in fem heltal i en vector och sortera dessa, genom att skriva ett predikat, utifrån deras avstånd från medianen, och skriv ut den sorterade vektorn.

Exempelvis för sekvensen 1, 3, 5, 6, 8 (där medianen är 5) så skulle utskriften bli 5, 6, 3, 8, 1.

Tips! För en sekvens på fem heltal så är alltid medianen i mitten i en sorterad lista.

Uppgift 5 – Queue och Stack

Uppgift 5.1

I en Queue (kö) så är det first-in, first-out som gäller. För att demonstrera hur en kö fungerar, lägg till talen 1, 2, 3, 4 till en kö genom att använda push. Skriv sedan en while loop som poppar och skriver ut talet längst fram tills kön är tom. Studera resultatet.

Uppgift 5.2

I en stack så är det istället first-in, last-out som gäller. Gör om samma program som ovan men använd en stack istället.

Uppgift 6 – Map

Uppgift 6.1

Skriv ett program som läser alfabetiska tecken från `stdin` till filslut (tips: skriv en text på fil och använd omdirigering). Programmet behöver endast kunna hantera det engelska alfabetet, dvs ej å, ä och ö. Programmet ska räkna hur många gånger ett givet alfabetiskt tecken förekom. Ingen skillnad görs på gemener och versaler. Blanksteg hoppas över. Programmet kan förutsätta att inga andra tecken än alfabetiska och blanksteg förekommer i inmatningen. Tecknen med tillhörande frekvens

ska lagras i en `std::map`. När inläsningen är klar ska innehållet i map:en skrivas ut.

Körexempel med texten ”hej alla glada” (utan citationstecken) läses in. Utmatningen blir:

```
a: 4
d: 1
e: 1
g: 1
h: 1
j: 1
l: 3
```

Inget speciellt behöver göras för att få en `map` att sortera stigande på nyckelelementet.

Uppgift 7 – Klasser och arv

Uppgift 7.1

Du ska här bygga på klassen `Animal` som du gjorde i *Övning 1*, *Uppgift 4*. Kopiera *Animal.h* och *Animal.cpp*.

- `Animal`
  Gör `say_hello` virtuell samt lägg till en rent virtuell (pure virtual) publik metod `sound`.
- Skriv en klass `Bird` som ärver `Animal`. När en `Bird` ”låter” (`sound`) ska den skriva ut sitt namn följt av ”kviddevitt! *picka*”.
- Skriv en klass `Cat` som ärver `Animal`. När en `Cat` ”låter” (`sound`) ska den skriva ut sitt namn följt av ”miuaw! *riva*”.
- Skriv en klass `Dog` som ärver `Animal`. När en `Dog` ”låter” (`sound`) ska den skriva ut sitt namn följt av ”vovv!”.

Instansiera varje klass ovan minst en gång. Dels som pekare till respektive typ, dels som vanlig typ. Låt var och en av objekten – pekare och inte pekare – säga hej (`say_hello`) samt ”låta” en gång var (`sound`).

Uppgift 7.2

Gör nu en lista (vector) som består av pekare till `Animal`. Men instansiera `Bird`, `Cat` och `Dog` i listan.

Exempel:

```
vector<*Animal> animals;
animals.push_back(new Bird("Beepo"));
animals.push_back(new Cat("Razer"));
animals.push_back(new Dog("Fido"));
```

Låt var och en av objekten i listan säga hej (`say_hello`) samt ”låta” en gång var (`sound`). Använd iteratorer och kom ihåg att en iterator är en pekare.

Får du någon skillnad mellan 7.1 och 7.2? Varför?

Laboration 2 – STL

Uppgift 1 – ”Romerska tal”

Läs in ett romerskt tal från standard inmatningsfil (`stdin`) i form av en sträng. Konvertera talet till ett vanligt heltal (`int`) och skriv ut resultatet.

Tips! De romerska siffrorna anges med bokstäverna I, V, X, L, C, D och M som står för 1, 5, 10, 50, 100, 500 respektive 1000. I ett romerskt tal gäller att om en romersk siffra P står omedelbart till vänster om en annan romersk siffra Q och om P betecknar ett mindre tal än Q så ska värdet av P subtraheras från det totala talet. Exempelvis är IV konverterat 4 och CXIV blir 114.

Uppgift 2 – ”Sortera romerska tal”

Fortsätt på programmet du skrev i uppgift 1. Modifiera programmet så att man börjar med att läsa in ett antal romerska tal (strängar) som alla ska läggas in i en `vector<string>`. Inläsningen avslutas då användaren skriver in bokstaven q (alternativt tills ett ogiltigt romerskt tal läses in). Denna `vector` ska sedan sorteras i nummerordning (lägst överst) genom att använda STL-algoritmen `sort` med ett binärt predikat. Skriv ut de romerska talen i den sorterade listan med hjälp av en iterator.

Uppgift 3 – ”Listor och dubbletter”

Skapa en kopia av programmet du skrev i Uppgift 2 och fortsätt på denna. Modifiera programmet så att du använder en `list<string>` istället för en `vector`. Nu ska dessutom en kontroll göras så att listan som skrivs ut enbart består av unika romerska tal.

Uppgift 4 – ”En Romersk Klass”

Skapa klassen `Roman`. Klassen ska ha:

- En konstruktor som tar ett romerskt tal som argument
- En andra konstruktor som tar en `unsigned` som argument
- En metod `dec` som returnerar det decimala talet i form av en `unsigned`
- En metod `rom` som returnerar det romerska talet (`string`)
- En klassmetod `count` som returnerar antalet romerska tal (instanser) som en `unsigned`

Gör ett litet ”spel” som låter en spelare konvertera tal. Spelaren ska konvertera mellan decimala heltal och romerska tal (åt båda håll). Om spelaren svarar rätt ges ett poäng. Spelet fortgår tills spelaren svarar fel. Uppnådda poäng ska skrivas ut för varje fråga. Talen ska vara större än noll och mindre än 5000. Fråga inte efter samma tal av samma typ två gånger.

Övning 3 – Strömmar

Uppgift 1 – hex

Skriv ett program som läser ett antal heltal från användaren och skriver ut deras hexadecimala representation. Heltalen kan skiljas åt antingen med mellanslag eller radbrytning - eller både och.

Programmet ska avslutas om något annat än heltal, radbrytning eller mellanslag skrivs in. Godtyckligt antal tal ska kunna anges under körning.

Exempel

```
"10 16 32" -> a 10 20
```

Uppgift 2 – ASCII

Skriv ett program som läser en textrad och skriver ut dess representation i ASCII. ASCII är de tal som representerar tecknen du skriver in. Avsluta inmatningen med radbrytning.

OBS: Beroende på språkinställningar så kan svenska tecken ge knasiga resultat.

Exempel

```
"ABC 123" -> 65 66 67 32 49 50 51
```

Uppgift 3 – cat

Skriv ett program som frågar efter en textfil och skriver ut den på skärmen.

TIPS: Funktionen `rdbuf()` i filströmmar kan göra uppgiften mycket enkel.

Uppgift 4 – Summera

Skriv ett program som läser in ett godtyckligt antal heltal från användaren och summerar dem. Talen ska skiljas åt med mellanslag. Radbrytning avslutar. Svaret ska skrivas ut på standard utmatningsfil (`stdout`).

Det är ok att anta att endast heltal ges från användaren.

Uppgift 5 – cat 2

Skriv ett program som ber om två filnamn. Programmet ska skapa en ny fil vars namn är de båda tidigare filernas namn sammansatta. Innehållet ska vara också vara en sammansättning.

Exempel

`fil1` innehåller `123` och `fil2` innehåller `ABC`. Programmet ska då skapa filen `fil1fil2` som innehåller `123ABC`.

Laboration 3 – Strömmar

Uppgift 1 – Sortera och läsa data

Skriv ett program som frågar om användaren vill sortera eller läsa data. Programmet ska sedan göra det användaren väljer och därefter avslutas.

- *Sortera data*
  Programmet frågar efter en textfil och sparar innehållet i en ny fil med samma namn och suffixet .sort tillagt (så a.txt blir a.txt.sort). Textfilen ska innehålla rader med heltal och det är ok att anta att filen alltid är korrekt - men glöm inte att kontrollera att den existerar. Programmet ska sortera raderna i nummerordning.
- *Läsa data*
  Programmet frågar efter en textfil och skriver ut den på skärmen.

Uppgift 2 – Filhantering

Skriv ett program som gör följande:

1. Frågar efter ett namn på en textfil.
2. Frågar efter ett ord utan mellanslag.
3. Skriver ut alla rader i filen som innehåller det angivna ordet.

Att ta hänsyn till om ordet är i rätt case (uppercase/lowercase) är upp till programmeraren. Glöm inte att kontrollera om filen existerar. Finns den inte så kan programmet antingen avslutas eller fråga igen.

Uppgift 3 – Stränghantering

Skriv ett program som ber om en sträng. Ignorera allt i strängen som inte är en siffra. Summera alla tal som är kvar i strängen. Observera att programmet måste hantera fallet med att strängen innehåller annat än siffror.

Exempel

```
"1 2 3" => 6
"1.1-2!4i20" => 28
```

Tips: strängar har många bra funktioner för att leta efter tecken.

Uppgift 4 – Roman II

Gör en kopia av `Roman` till denna uppgift. Skriv ett program som läser in romerska tal och decimala heltal (blandat) från standard inmatningsfil (`stdin`) och skriver ut de konverterade talen på standard utmatningsfil (`stdout`). Talen är separerade med vittecknen mellanrumstecken eller radbrytning. Använd `Roman` för konverteringen.

Felaktiga poster i inläsningen ska ge utskrift av felaktig post till standard felkodsfil (`stderr`) med lämpligt felmeddelande. Utskriften ska inte vara buffrad.

Flaggor

Gör så att man kan ange flaggan `--sum` på kommandoraden för att få endast summan av talen utskrivna på `stdout`.

Gör så att man kan ange flaggan `--sum-separately` på kommandoraden för att få summan av alla decimala heltal (som ett decimalt heltal) följt av summan av alla romerska heltal (som ett romerskt heltal) – båda på samma rad, separerade med ett mellanrumstecken.

Övning 4 – minne

Uppgift 1 – pekare

Skriv ett program som skapar en pekare till en integer, ger variabeln ett värde, samt slutligen tar bort pekaren och kontrollerar att den är borttagen.

Uppgift 2 – summera int-pekare

Skriv ett program som skapar två pekare till olika heltal (`int`). Använd standard inmatningsfil för att ge värden till dessa. Summera dem samt skriv ut resultatet.

Uppgift 3 – referera pekare

Är det skillnad på att skriva `*&i` och `&*i` (där i är en pekare till en `int`)? Resonera dig fram till ett svar och skriv sedan ett program som testar om du har rätt.

Uppgift 4 – pekare och referenser

Undersök den givna koden nedanför och försök resonera dig till vad utskriften blir. Skapa sedan ett program som kör koden och se om du har rätt.

```
int val;
int* p1;
int* p2;

val= 1;
p1 = new int;
*p1 = 2;
p2 = new int;
*p2 = 3;

std::cout << "p1 är " << *p1
        << "\np2 är " << *p2
        << "\nval är " << val
        << std::endl;

val = *p1;
p2 = &val;

std::cout << "p1 är " << *p1
        << "\np2 är " << *p2
        << "\nval är " << val
        << std::endl;
```

Uppgift 5 – pekare och referenser 2

Om du precis före den sista `std::cout`-raden i koden i uppgiften innan lägger till:

```
(*p2)++;
```

Vad kommer utskriften bli då?

Uppgift 6 – increase

Givet är programmet nedan:

```
#include <iostream>

void increase(int v) {
      ++v;
}

int main() {
    int val = 3;
    increase(val);
    std::cout << "3 + 1 = " << val << std::endl;
    return 0;
}
```

Om du kör ovanstående program så kommer du att märka att det inte fungerar som det är tänkt. Modifiera funktionen `increase` (koden i `main`-funktionen får inte modifieras) så att den genom användandet av referenser gör att programmet ger rätt utskrift.

Uppgift 7 – *increase

Gör om uppgiften ovan men använd istället pekare. Du måste även lägga till ett tecken till `main`-funktionen för att det ska fungera.

Uppgift 8 – vector

Skapa en pekare till en `vector` innehållande `int`. Lägg till talsekvensen 4, 5, 3, 1, 2 genom att använda `push_back`. Sortera sedan talen (använd STL-funktionen `sort`) och skriv ut hela sekvensen med hjälp av en iterator. (Tips! Samma sak som i laborationen om STL fast med pekare!)

Laboration 4 – Fordon med klass

Uppgift 1 – Vehicle, Taxi

Vehicle.h

```
#ifndef VEHICLE_H_
#define VEHICLE_H_

#include <iostream>
#include <string>

class Vehicle {
public:
     Vehicle(const std::string& reg_number,
          const std::string& color,
          const int number_of_doors);
     virtual ~Vehicle();

     void set_color(const std::string& color);
     std::string get_color() const;

     int get_number_of_doors() const;

     std::string get_reg_number() const;
     virtual std::string get_class_name() const;

     virtual void print(std::ostream& os) const;

     static int get_number_of_objects();

     friend std::ostream& operator<<(
          std::ostream& os,
          const Vehicle& rhs);
private:
     const std::string vehicle_reg_number;
     std::string vehicle_color;
     const int vehicle_number_of_doors;
};

#endif /* #ifndef VEHICLE_H_ */
```

Skriv en tillhörande implementationsfil för klassen `Vehicle` som fullständigt definierar den. Lägg även till en statisk variabel som håller reda på antal existerande `Vehicle`-objekt.

Taxi.h

```
#ifndef TAXI_H_
#define TAXI_H_

#include <iostream>
#include <string>

#include "Vehicle.h"

class Taxi: public Vehicle {
public:
     Taxi(const std::string& reg_number);

     bool has_customers() const;
     void set_has_customers(const bool flag);

     // Virtual in base class (vehicle)
     std::string get_class_name() const;

     // Virtual in base class (vehicle)
     void print(std::ostream& os) const;

     friend std::ostream& operator<<(
          std::ostream& os,
          const Taxi& rhs);

private:
     bool customers;
};

#endif /* #ifndef TAXI_H_ */
```

Utgå från definitionen i filen `Taxi.h` och modifiera denna så att klassen `Taxi` ärver från klassen `Vehicle` och skapa en implementationsfil för denna klass där du skriver koden för funktionerna.

En taxi är alltid gul och har fyra dörrar. Det ska också finnas en datamedlem av typen `bool` som håller reda på om taxin är ledig eller ej.

I samtliga klasser så ska `operator<<` anropa `print` som ska sköta all utskrivning.

Ett färdigt program finns för att testa din kod i filen `test_vehicle.cpp`.

Utskriften ska bli:

```
Number of objects: 0

Created two vehicles.

Number of objects: 2

Vehicle:
Registration number: B3A2
Number of doors: 4
Color: blue

Taxi:
Registration number: B3A1
Number of doors: 4
Color: yellow
Has no customers

Taxi picked up a customer.

Taxi:
Registration number: B3A1
Number of doors: 4
Color: yellow
Has customers

Deleted two vehicles.

Number of objects: 0
```

Uppgift 2 – Limo

Gör nu om klassen `Vehicle` till en abstrakt klass. Gör dess `print`-funktion rent virtuell (pure virtual).

Skapa även en klass `Limo` som ska ärva från klassen `Taxi`. `Limo` ska ha samma funktionalitet som `Taxi`-klassen. En `Limo` är alltid vit och har fyra dörrar.

Du ska inte behöva ändra åtkomsträttigheter för medlemsvariablerna i `Taxi` eller `Vehicle` för att åstadkomma detta och du ska inte heller behöva lägga till några fler konstruktorer.

Ett färdigt program finns för att testa din kod i filen `test_limo.cpp`.

Utskriften ska bli:

```
Number of objects: 0

Created two vehicles.

Number of objects: 2

Limo:
Registration number: B3A3
Number of doors: 4
Color: white
Has no customers

Taxi:
Registration number: B3A4
Number of doors: 4
Color: yellow
Has no customers

Deleted two vehicles.

Number of objects: 0
```

Uppgift 3 – Car_Park

Skapa en fristående klass `Car_Park` som håller reda på hur många och vilka fordon som är parkerade där. Fordonen ska lagras i en lista som har pekare till basklassen `Vehicle`, dvs en `list<Vehicle *>`. Konstruktorn för `Car_Park` ska ta storleken på parkeringshuset som argument.

Skriv funktionen `park` som ska parkera en `Vehicle*`, som skickas med som argument till funktionen (se testprogrammet), i parkeringshuset om plats finns. Funktionen `unpark` ska ta bort ett fordon med specificerat registreringsnummer ur listan. Funktionerna `park` och `unpark` ska ge feedback i form av utskrifter, se testprogrammet för detaljer.

Klassen ska även ha en funktion `print` som ska skriva ut en lista på de fordon som är parkerade i parkeringshuset. `print` ska ta en `ostream`-referens som argument och skriva ut på denna ström.

Ett färdigt program finns för att testa din kod, `test_car_park.cpp`.

Utskriften ska bli:

```
Parked a Taxi
Parked a Taxi
Not enough room to park the Limo

The following vehicles are in the carpark:
Taxi:
Registration number: T1
Number of doors: 4
Color: yellow
Has no customers

Taxi:
Registration number: T2
Number of doors: 4
Color: yellow
Has no customers

T1 left the carpark.
Parked a Limo

The following vehicles are in the carpark:
Taxi:
Registration number: T2
Number of doors: 4
Color: yellow
Has no customers

Limo:
Registration number: L1
Number of doors: 4
Color: white
Has no customers
```

Övning 5 – Klasser i C++

Uppgift 1

Denna uppgift går ut på att implementera en klass för att representera datum. Du ska med hjälp av att titta på headerfilen (`.hpp`) skriva en källkodsfil (`.cpp`) som implementerar alla klassens medlemsfunktioner. Glöm inte att inkludera "`Date.hpp`" i din "`Date.cpp`". För varje medlemsfunktion finns en kommentar som anger vad den ska utföra. Gör sedan ett program som testar att klassen fungerar som den ska. Testprogrammet ska ligga på en separat fil `main.cpp`.

Date.hpp

```
class Date {
public:
     // Initiera medlemsvariablerna med dessa parametrar.
     Date(int year, int month, int day);

     // Dessa funktioner returnerar värdet av sin variabel.
     // Ex: ett_datum.get_year() returnerar my_year hos
     // objektet ett_datum.
     int get_year() const;
     int get_month() const;
     int get_day() const;

     // Dessa funktioner ska sätta sin variabel till värdet som
     // skickas med.
     // Ex) ett_datum.set_year(2007) sätter my_year till 2007
     // hos objektet.
     void set_year(int year);
     void set_month(int month);
     void set_day(int day);

private:
     int my_year;
     int my_month;
     int my_day;

};
```

Notis om filändelsen `.hpp`: C++-programmerare som skapar sina egna headerfiler ger dem ändelsen `.h` eller `.hpp` i vanliga fall. `.hpp` signalerar att det är en C++ header och inte en C header – man får alltså direkt en ledtråd om språket i filen utan att ens öppna den. Filändelsen för en headerfil kan vara avgörande för hur en editor fontifierar filen. Programmerar man i både C och C++ och använder samma editor är det bäst att använda olika filändelser.

Uppgift 2

Undersök nedanstående program och resonera dig fram till vad utskriften kommer att bli. Skapa sedan ett program som testar koden och se om du hade rätt.

```
#include <iostream>

class A {
public:
   A() { std::cout << "A" << std::endl; }
   ~A() { std::cout << "~A" << std::endl; }
};

class B : public A {
public:
   B() { std::cout << "B" << std::endl; }
   ~B() { std::cout << "~B" << std::endl; }
};

int main() {
   A *a = new A;
   B *b = new B;

   delete a;
   delete b;
}
```

Uppgift 3

Hur löser man problemet nedan? Du vill ha en A i en B och en B i en A...

A.h

```
#ifndef A_H_
#define A_H_
#include "B.h"

class A {
public:
     B b;
};
#endif /* A_H_ */
```

B.h

```
#ifndef B_H_
#define B_H_
#include "A.h"

class B {
public:
     A a;
};
#endif /* B_H_ */
```

Laboration 5 – ”tuff sa tåget”

Uppgift 1

Denna uppgift är lite mer omfattande och det är viktigt att läsa igenom uppgiftsbeskrivningen noggrant. Ni måste även grundligt gå igenom tillhörande dokument som beskriver de algoritmer som ska användas. Försök inte skriva hela programmet på en gång utan dela upp uppgiften i deluppgifter som ni kan angripa en i sänder. Man vill snabbt ha ett program som går att kompilera som steg för steg byggs på med en noga testad funktion, klass etcetera tills uppgiften är löst i sin helhet. Divide and conquer!

Uppgiften går ut på att skriva en ”miniräknare”. Miniräknaren behöver bara kunna hantera de fyra räknesätten (+, -, * och /) och den behöver dessutom endast klara av att räkna med positiva heltal (>= 0). Givet ett matematisk uttryck som uppfyller reglerna ovan, exempelvis, ”1 + 2 * 3”, hur skriver man då ett program som kan beräkna sådana uttryck? Jo, man kan använda den så kallade järnvägsalgoritmen som gör om ”1 + 2 * 3”, som är i skriven i infixform, till postfixform och sedan beräknar med hjälp av en stackbaserad algoritm. Konverteringen infix till postfix och själva beräkningen av postfixuttrycket finns beskrivet i detalj i en separat pdf.

Programmet ska vara uppbyggt genom ett antal klasser.

Calculator

Denna klass ska instantieras av `main()` som sedan ska anropa dess publika medlemsfunktion `read_and_evaluate_expressions()`. Denna funktion ska läsa infixuttryck radvis från `stdin` till EOF. Varje infixuttryck som lästs in ska omvandlas till postfixform, beräknas och skrivas ut. Utskriften ska inkludera uttrycket i både infix- och postfixform och vad uttrycket evaluerades till.

Om ett ogiltigt uttryck påträffas (mer om det längre ned) ska en felutskrift visas och programmet ska sedan behandla nästa uttryck.

Postfix

Klassen `Postfix` är den klass som gör grovjobbet i programmet och det är den som `Calculator` ska använda sig av för att omvandla och beräkna uttryck. `Postfix` ska ha två publika medlemsfunktioner: `infix_to_postfix()` och `evaluate()`. Funktionen `infix_to_postfix()` ska ta ett infixuttryck i strängform och returnera det i postfixform lagrat i en `deque<string>`. Notera! Postfixformen som returneras ska lagras i en `deque<string>`, inte i en `string` (men infixuttrycket funktionen tar emot är lagrat i en `string`). En `deque` påminner om en `vector` men har en del medlemsfunktioner som gör evalueringen enkel. `evaluate()` ska ta ett postfixuttryck i form av en `deque<string>`, beräkna och ge tillbaka svaret som en `double`.

Token_Handler, Operand_Handler och Operator_Handler

`Postfix`-klassen behöver ett sätt att ta reda på om en given token är en operand eller en operator, och det arbetet ska skötas av klasserna `Operand_Handler` och `Operator_Handler`.

`Token_Handler` är en abstrakt basklass till dessa klasser och har endast en (publik) medlemsfunktion: den rent virtuella `is_of_type()`. Denna ska definieras av subklasserna `Operand_Handler` och `Operator_Handler`. Den ska returnera en `bool` och tar en token i form av en `string` och kontrollerar att denna tokens typ stämmer överens med subklassen för vilken den anropades

- `Operand_Handler::is_of_type()`
  ska returnera `true` om token är en operand, annars `false`
- `Operator_Handler::is_of_type()`
  ska returnera `true` om token är en operator, annars `false`

`Operand_Handler` blir en mycket enkel klass eftersom den bara behöver kunna avgöra om en given token är en operand eller ej. `Operator_Handler` ska kunna avgöra om en given token är en operator eller ej men även kunna ta reda på en operators prioritet och blir inte lika trivial som sin systerklass.

Vill man komplettera med hjälpfunktioner till klasserna går det bra men de ska vara privata. Statiska datamedlemmar och statiska medlemsfunktioner kan också vara motiverade. Faktorisera egna funktioner väl för att underlätta felsökning och läsbarhet.

Sammanfattning över vilka klasser som skapas och deras givna publika gränssnitt:

```
Calculator:
void read_and_evaluate_expressions()

Postfix:
deque<string> infix_to_postfix(const string&)
double evaluate(deque<string>)

Token_Handler (abstrakt basklass):
bool is_of_type(const string&)

Operator_Handler (subklass till Token_Handler):
bool is_of_type(const string&)

Operator_Handler (subklass till Token_Handler):
bool is_of_type(const string&)
```

Varje klass ska ha sin egen headerfil och implementationsfil (`Token_Handler` behöver dock ingen implementationsfil) och kom ihåg att `using`-direktiv inte är tillåtet att använda i headerfiler.

Felhantering

Programmet ska kunna hantera följande fel i infixuttrycken:

1. Ogiltigt tecken: tecken som inte är en giltig operand eller operator.
2. En operand ska finnas på båda sidorna om en operator.
3. En operator ska alltid följa på en operand (uttryck som består av en enda operand, t ex ”1” ska dock tillåtas. Se exempelkörningen nedan för exempel på tillåtna och otillåtna uttryck).

Så fort något av detta fel påträffas i ett uttryck ska behandlingen av det uttrycket avbrytas och

programmet gå vidare till nästa uttryck. Exakt hur avbryter man då? Jo, genom att använda så kallade *exceptions*. Eftersom undantag (exceptions) inte gås igenom på föreläsningarna kommer mycket stöd att ges.

Felmeddelande ska skrivas ut till standard felkodsfil (`stderr`).

Här är ett komplett exempelprogram:

```
#include <iostream>
#include <stdexcept> // För runtime_error

double divide(double t, double n) {
     if (n == 0) {
          throw std::runtime_error("You may not divide by 0!");
     }

     return t / n;
}

int main() {
     try {
          std::cout << divide(1, 0) << std::endl;
          // I detta block kan man ha flera satser
     } catch (const std::runtime_error& e) {
          std::cerr << e.what() << std::endl;
          // ... och även i detta
     }
}
```

I detta exempelprogram har vi en funktion `divide()` som tar två doubles, dividerar dem och returnerar resultatet. Men vi vill inte dividera med `0` utan väljer att kasta ett undantag ifall nämnaren är `0`. När ett undantag har kastas så nystas hela anropskedjan upp tills någon funktion fångar undantaget. I vårt fall är det `main()` som fångar undantaget (notera `try`/`catch`-blocket). När `main()` fångar ett eventuellt undantag skriver den ut den feltext som gavs. Om ingen fångar undantaget så kommer programmet att krascha med ett *unhandled exception error*. Det är starkt rekommenderat att ni leker med detta program om ni är osäkra. Det finns även mycket information i kursböckerna. I programmet ni ska skriva är det `Calculator::read_and_evaluate_expressions()` som ska ha `try`/`catch`-blocket (kapsla in anropen till `Postfix::infix_to_postfix()` och `Postfix::evaluate()` i `try`-blocket). `Catch`-blocket ska skriva ut felbeskrivningen genom att anropa `what()` på undantagsobjektet som i exemplet. Dessutom ska det uttryck som genererade undantaget skrivas ut. Om man i stället lägger `try`/`catch` i `main()` kommer programmet inte behandla fler uttryck så fort ett undantag har kastas. Det vill vi inte. Vi vill gå till nästa infixuttryck då och därför är det `Calculator`-klassen som ska fånga undantagen.

Prioritetstabeller

Följande prioritetstabell för de operatorer programmet ska ha stöd för ska användas:

```
+    1
-    1
*    2
/    2
```

Notera att pdf:en som beskriver omvandling infix-postfix och evaluering av postfixuttryck även talar om parenteser och symboliska variabler, bland annat. Er miniräknare ska inte ha stöd för sånt.

Inläsning och körning

Uttrycken ska behandlas radvis som det nämndes ovan och då är `getline()` en bra kandidat för att läsa ett uttryck. Programmet ska förutsätta att det finns whitespace mellan varje token i ett infixuttryck, vilket förenklar bearbetningen avsevärt.

Det är lämpligt att man skriver en textfil med ett antal infixuttryck (radvis, både giltiga och ogiltiga) och låter programmet läsa den filen med hjälp av omdirigering i skalet. Här följer en serie uttryck som ni kan lägga i en textfil (komplettera gärna med egna) och kontrollera att ni får samma utmatning:

```
1 + 2
1 + 2 * 3
1 / 2 - 9
1 * 2 - 4 / 2 + 0
99 / 128
1 +
+ 1
1 + + 2
+
1
1a
1 2
```

Utmatning

```
The infix expression 1 + 2 is 1 2 + in postfix form and evaluates
to 3

The infix expression 1 + 2 * 3 is 1 2 3 * + in postfix form and
evaluates to 7

The infix expression 1 / 2 - 9 is 1 2 / 9 - in postfix form and
evaluates to -8.5

The infix expression 1 * 2 - 4 / 2 + 0 is 1 2 * 4 2 / - 0 + in
postfix form and evaluates to 0

The infix expression 99 / 128 is 99 128 / in postfix form and
evaluates to 0.773438

Caught exception for expression: 1 +
First error was: There must be an operand after each operator!

Caught exception for expression: + 1
First error was: There must be an operand before each operator!

Caught exception for expression: 1 + + 2
First error was: There must be an operand before each operator!

Caught exception for expression: +
First error was: There must be an operand before each operator!

The infix expression 1 is 1 in postfix form and evaluates to 1

Caught exception for expression: 1a
First error was: Exception in evaluate(): Encountered a symbol
"1a" that is neither a valid operator nor a valid operand.

Caught exception for expression: 1 2
First error was: There must be an operator between two operands!
```