Datorlaboration 1

Denna laboration introducerar,

Bygg- och Exekveringsprocessen
: Kompilering, länkning, header-/source-filer, redirect av strömmar, Makefiler.

Grundläggnde C++-koncept
: Input-/output-strömmar, klasser, metodargument.

Den interna kompileringen i Visual C++ sker med hjälp av verktygen `cl.exe` och `link.exe` (med flera). I denna laboration kommer dessa verktyg att användas direkt via command-line för att bygga ett antal mindre C++-program. Att kompilera (och assemblera, länka, etc) via command-line går utmärkt för mindre projekt och ger god insyn byggprocessen.

Laborationen ger även en kort introduktion till NMAKE, Windows' motsvarighet till UNIX's *make*, som är ett verktyg för så kallde *Makefiler*. Makefiler är en slags instruktionsfil som beskriver hur kompileringen ska gå till, hur filer relaterar till varandra och vilka targets som ska genereras.

Command-line kompilering

Command-line kompilering sker via en särskild konsol, *Developer Command Prompt*, som är förinställd med diverse nödvändiga environment variabler. Öppna den i Windows via:

```
Start > Apps > Visual Studio 2015 > Developer Command Prompt
```

Gå till `lab1/hello`-katalogen och öppna filen `hello.cpp`. Exempelvis med notepad:

```
notepad hello.cpp
```

Denna fil innehåller ett enkelt *Hello World!*-program som vi nu ska kompilera.

Syntaxen för `cl.exe` är

```
CL [option...] file... [option | file]...
     [lib...] [@command-file] [/link link-opt...]
```

Här representerar `option` kompilatorflaggor (exempelvis optimeringsgrad); `file` och `lib` avser referenser till källfiler, obj-filer (för-kompilerade filer) eller bibliotek; och `link-opt` är länkinstruktioner. `command-file` är ett alternativ för att spara inställningar i en separat fil. Ett minimalt kommando för att kompilera källfilen `foo.cpp` är

```
cl /EHsc foo.cpp
```

Detta kommando kompilerar och länkar den angivna filen till en körbar fil med namnet `foo.exe`. Exe-filen kan döpas om genom att istället ange

```
cl /EHsc /Fe:foobar.exe foo.cpp
```

Kompilering är att skapa filer med maskinkod (*obj*) och **länkning** är att koppla samman *obj*-filer till ett *target*, exempelvis en exekverbar fil (*exe*) eller ett bibliotek (*lib*, *dll*).

Urval av kompilatorflaggor:

`/Fe`	Ange ett specifikt target-namn
`/EHsc`	Aktivera säker C++ felhantering (denna flagga ska vi alltid ha med)
`/c`	Kompilera utan att länka
`/w`	Ignorera alla varningar
`/Wall`	Ange alla varningar
`/Wx`	Behandla varningar som fel
`/O1`	Optimera för liten filstorlek
`/O2`	Optimera för prestanda
`/I include`	lägg till sökväg till include-katalogen `include`
`/LD`	Kompilera till ett dynamiskt länkat bibliotek (DLL), istället för en körbar fil.

Kompilering med separat länkning sker alltså enligt t.ex.,

```
cl /EHsc /c foo.cpp

link /EHsc /out:foo.exe foo.obj
```

Då vi har flera källfiler anges dessa helt enkelt efter varandra,

```
cl /Fe:myapp.exe main.cpp myclass.cpp
```

Uppgift 1: Hello World

Kompilera `hello.cpp` med kommandot:

```
cl /EHsc hello.cpp
```

Vilka nya filer genererades?
Exekvera programmet med:

```
hello
```

Vad skrevs ut, och var?
Syns något fel i utskriften? Rätta i så fall felet och gör om processen.

Uppgift 2: Separat Länkning

Radera de genererade filerna. Kompilera `hello.cpp` igen, nu med separat länkning.

Undersök vilka filer som skapas efter kompilering respektive länkning.

Uppgift 3: Inparametrar

Utöka main-metoden i `hello.cpp` så att den kan ta emot inparametrar.

Gå igenom (iterera) arrayen av inparametrar och skriv dem till `std::cout`.

Låt exempelvis exekvering med följande inparametrar,

```
hello Bjarne Stroustrup
```

ge föjande utskrift:

```
Hello World! Nice to see you, Bjarne Stroustrup!
```

Kompilera och testa.

Kalkylator

Indata till `std::cout` skriver symboler till programmets output-ström, som per default dirigeras till konsolen. Programmet har också en input-ström som läses via `std::cin`. Också inströmmen är, per default, dirigerad till konsolen, vilket betyder att användaren ges en prompt att ange valfri indata så fort `std::cin` används.

In-strömmen kan läsas på olika sätt. Med `std::cin.get()` läses ett enda tecken. En sekvens av tecken (en *token*) kan också tolkas direkt som en variabel:

```
std::cin >> val;
```

där `val` är en variabel av lämplig typ, exempelvis `int` eller `float`. Flera tokens kan också läsas i sekvens:

```
std::cin >> val1 >> val2;
```

För en ström av indata av okänd längd behövs en loop:

```
while(std::cin >> val)
    // take care of 'val'
```

Loopen avbryts då strömmen `std::cin` evalueras till `false`, vilket kan ske av olika skäl. Ett är att innehållet i `std::cin` inte kan konverteras till typen för `val`. Ett annat är att `std::cin` har påträffat en end-of-file (EOF) karaktär. EOF inträffar då en fil tar slut, eller då användaren matar in Ctrl-x (Ctrl-d i UNIX). Se *Lippman* för mer information.

Uppgift 4: `std::cin`-kalkylator

Skapa en katalog `lab1/sum` och skriv ett program, `sum.cpp`, som hämtar in tal från användaren/konsolen och sedan skriver ut summan när inmatningen är klar. Talen hämtas in via `std::cin` och antalet tal ska vara godtyckligt.

Användaren avslutar inmatningen genom att ange EOF (Ctrl-x).

Både `std::cout` och `std::cin` kan omdirigeras från eller till andra mål än konsolen med hälp av shell-kommandon. Redirects påverkar inte själva programmet, utan utförs av den shell programmet exekveras ifrån. Filer med serier av shell-kommandon kallas för shell-script eller batch-script (.bat).

Uppgift 5: Output-redirect till fil

Dirigera output-strömmen för `sum.exe` från konsolen till en fil:

```
sum > sum.txt
```

Kör programmet och undersök den genererade filen.

Shell-operatorn `>` dirigerar till en ny fil varje gång. Använd `>>` för att läga till (append) i en befintlig fil.

Uppgift 6: Input-redirect från fil

Skapa en text-fil, `terms.txt`, med en serie tal separerade med mellanslag. Talen kan vara på samma rad eller på flera rader. Dirigera input-strömmen till `sum.exe`, från konsolen till att komma fån `terms.txt`:

```
sum < terms.txt
```

Var hamnar output-strömmen? Vad innehåller den?

Uppgift 7: Input/output-redirect

Gör nu redirects av både input- och output-strömmarna. Detta betyder att programmet `sum.cpp` läser in tal från en fil, summerar, och skriver ut resultatet i en annan fil.

```
(sum < numbers.txt) > sum.txt
```

Verifiera att filen `sum.txt` innehåller vad den ska.

Strömmar kan också vidarebefordras *mellan program*. En sådan struktur kallas för *pipe* och skrivs som

```
prog1 | prog2
```

I praktiken hamnar det som skrivs till `std::cout` i `prog1`, i `std::cin` för `prog2`.

Polynom-Solver

Ett reguljärt andragradspolynom har formen

$$y(x) = ax^2 + bx + c, \tag{1}$$

där är a, b och c är polynomets koefficienter, som kan antas vara heltal skilda från noll. Rötterna till Ekvation **??** ges av x för $y(x) = 0$, där

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \tag{2}$$

Uttrycket inuti kvardratroten kallas för diskriminant, $d = b^2 - 4ac$. En rotlösare betraktar följande utfall för diskriminanten:

- Om $d < 0$ så finns två imaginära rötter (eftersom kvadratroten ur ett negativt tal ger ett imaginärt tal). Dessa ignorerar vi.
- Om $d = 0$ så finns en (1) reell rot (en dubbelrot): $-b/2a$ (Ekvation **??**).
- Om $d > 0$ så finns två reella rötter, enligt Ekvation **??**.

Uppgift 8: Polynom-klassen

Definiera metoderna `eval` och `findRoots` i källfilen `lab1/poly/poly2.cpp`. Metoden `eval` ska ge värdet av Ekvation **??** för ett visst x, och metoden `findRoots` ska skriva ut (eventuella) rötter enligt beskrivningen ovan.

Skapa flera polynom i main-metoden med olika koefficienter, inklusive dem i Figur **??**. Gör testanrop till både `eval` och `findRoots`.

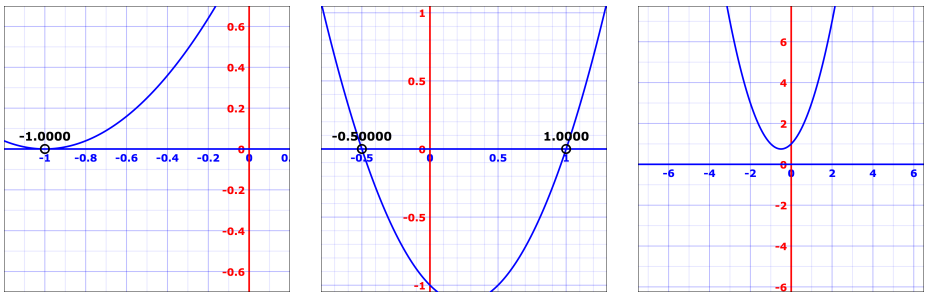
Tips: En funktion `sqrt` finns i standardbiblioteket `<cmath>`.

Uppgift 9: Returnera rötter till anroparen

Skriv om `findRoots` så att metoden returnerar information om rötterna till main-metoden. Utskrift ska nu ske i main-metoden istället för i `findRoots`.

Lägg till lämpliga *by-reference* argument till `findRoots` för data som ska returneras. Information som behöver returneras är: antalet rötter, samt eventuella värden av dessa.

Verifiera erhållna rötter genom att anropa `eval` med varje rot som argument. Vad bör `eval` teoretiskt returnera om den ges en rot som argument?



$y(x) = x^2 + 2x + 1$
$x_1 = x_2 = -1$

$y(x) = 2x^2 - x - 1$
$x_1 = -0.5$
$x_2 = 1$

$y(x) = x^2 + x + 1$
$x_1, x_2 \in \mathbb{C}$

Figur 1: Exempel på andragradspolynom med rötter. Vänster: En dubbelrot. Mitten: Två rötter. Höger: Ingen reell rot. (Diagram: `https://www.mathsisfun.com`)

Tips: Utskrifter med formatet `4.76837e-07` betyder *grundpotensform*. I detta fall motsvarande $4.76837 \cdot 10^{-7}$ (det vill säga ett väldigt litet tal – så litet att vi betraktar det som noll).

Att tänka på: ändringar i en metods signatur måste göras identiskt i både .h-filen och cpp-filen (annars kan inte länkaren matcha ihop metodens definition med dess deklaration).

Uppgift 10: Koefficienter via `std::cin`

Utöka main-metoden i `polysolver.cpp` så att koefficienter läses in från användaren/konsolen.

Hämta in koefficienter tills EOF påträffas (användaren matar in Ctrl-x), och betrakta varje sekvens av tre tal som koefficenter för ett polynom. Skapa polynomet, hämta dess rötter och skriv ut resultatet. Fortsätt läsa in koefficienter, skapa ett nytt polynom, o.s.v..

Uppgift 11: Koefficienter via redirect

Filen `coeffs.txt` innehåller koefficienter för fyra polynom (det går bra att lägga till fler!).

Gör det möjligt för `polysolver.cpp` att både hämta koefficienter från en fil, `coeffs.txt`, samt skriva ut rötterna till en annan fil, `roots.txt`. En exekvering kan se ut som följer:

```
(polysolver < coeffs.txt) > roots.txt
```

Säkerställ att hela flödet av inläsning, rotlösning och utskrift fungerar som avsett. Notera att en EOF kommer att påträffas per automatik i slutet av filen, vilket ska leda till att inläsningen avbryts.
Notera återigen i utskriften, d.v.s. i `roots.txt`, värdet av `eval` för varje rot. Försök förklara varför värdet inte alltid är noll.

För koefficienter som inte är skilda från noll krävs ytterligare test för att hålla rotlösaren robust. Då $a = 0$ uppstår exempelvis en singularitet i Ekvation **??**. Betrakta uttrycket i Ekvation **??** och fundera över hur dessa fall kan hanteras. Vad är ordningen på polynomet om $a = 0$?

Uppgift 12: Nollkoefficienter (Valfri)

Utöka `findRoots` för att kunna hantera koefficienter som är noll.

Ännu ett steg mot en robustare rotlösare är att tillåta koefficienter som inte är heltal. Detta kan skapa problem då decimaltal inte alltid kan representeras exakt med befintlig standard för flyttal (IEEE 754).

Betrakta följande kontraintuitiva resultat, där

```
std::cout << std::setprecision(10) << 0.1f << std::endl;
```

ger utskriften:

```
0.1000000015
```

Även ett "snällt" tal som 0.1 är alltså inte exakt representerat av en 32-bits `float`. Detta gör att man måste vara väldigt försiktig med likhetstest när det kommer till flyttal. Även likhet med noll, som i vårt fall med d. Lösningen är att inte jämföra med exakt noll, utan med ett litet tal, ett epsilon ϵ. Istället för $d = 0$ testar man alltså $|d| < \epsilon$, där $|d|$ betyder absolutvärde. Epsilon kan sättas till t.ex. $\epsilon = 10^{-7}$.

Uppgift 13: Flyttalskoefficienter (Valfri)

Utöka `findRoots` för att kunna hantera koefficienter som är decimaltal.

Makefiler

Programbygge enbart via command-line blir snabbt omständigt när antalet filer, kompilatorflaggor och targets växer. Makefiler gör det möjligt att lägga samman olika kompileringsalternativ i en enda fil, en *makefile*. Makefilen tolkas av ett särskilt program, som i sin tur anropar kompilator, länkare etc. Makefilen kan också innehålla shell-instruktioner som t.ex. skapar kataloger eller flyttar filer.

Kompileringsprocessen kan abstraheras ytterligare och göras helt oberoende av platform, arkitektur och kompilator. *CMake* är ett system som skapar makefiles dynamiskt, beroende på den aktuella maskinens konfiguration. Den här typen av system är särskilt viktiga för distribution av platformsoberoende kod, och det är inte ovanligt att mjukvarupaket från t.ex. Github levereras med enbart källkod och en *CMake*-fil. På så sätt slipper utvecklaren ta fram och underhålla releaser för alla tänkbara konfigurationer.

Makefile-tolkaren vi använder är Microsoft's NMAKE. För att bygga ett projekt i en katalog där det finns en makefile, använd kommandot:

```
nmake all
```

Här fungerar *all* som en generell bygginstruktion, ett *pseudo-target*, som t.ex. bygger en körbar fil plus allting den beror på (i omvänd ordning). Ett annat vanligt pseudo-target är *clean*, som i princip reverserar processen genom att städa bort byggda targets, intermediate-filer och kataloger:

```
nmake clean
```

Uppgift 13: Enkel Makefile

Bygg programmet i katalogen `lab1/makefile_hello` med hjälp av NMAKE.

Notera vilka filer som skapas. Bygg sedan med *clean*, och notera igen vad som händer.

Undersök makefilen och försök förstå dess struktur. Ta hjälp av internetkällor om så behövs.

Uppgift 14: Makefile med underkataloger

Gå till katalogen `lab1/makefile_project`. Notera katalogstruktur med innehåll. Detta representerar hur kod ofta organiseras i större projekt (detta projekt råkar vara litet och trivialt).

Bygg med NMAKE och studera effekten. Undersök makefilen.