Anders Gidenstam
Karl Jansson

Laboration #1: Rekursion

Deadline: 2015-04-21

De övriga laborationerna i kursen Data- och programstrukturer utgörs av enskilda program som är relativt stora. För att ni skall få lite uppvärmning så består den första laborationen av 4 mindre problem varav inget kräver mer än en sida kod. Problemen kommer att tvinga er att tänka rekursivt och de flesta är tagna direkt ur boken.

Den här laborationen består i att skriva flera mindre rekursiva program. Det betyder dock inte att ni skall vänta med att börja till sista minuten – rekursiva funktioner är ofta relativt korta men rekursion kan vara lurigt i början.

Problemen finns återgivna nedan tillsammans med en del ytterligare kommentarer och instruktioner. På Ping-Pong finns de filer och bibliotek som krävs för laborationens genomförande.

Mer konkret finns följande filer som ni behöver för denna laboration på Ping-Pong:
PCLibs20xx.zip: Innehåller PCLibs och MazeLib för Visual Studio 20xx i katalogen lib och samtliga .h filer, d.v.s. även graphics.h och mazelib.h, i katalogen include. Observera att PCLibs kan vara utökat jämfört med förra kursen så använd denna version.

Exefiler.zip: Körbara filer för att visa önskad funktionalitet. I detta bibliotek ligger även labyrintfilerna till uppgift 4.

maze.c: Startfilen för uppgift 4.

Checklista för att använda Roberts' bibliotek i Visual Studio:

- I Project->Properties: C/C++->General
  Lägg in sökvägen till Roberts' header filer i "Additional Include Directories".
  T.ex. '..\..\include' om du lagt include biblioteket utanför lösningens bibliotek.
- I Project->Properties: Linker->General
  Lägg in sökvägen till biblioteket där PCLibs20xx.lib ligger i "Additional Library Directories".
  T.ex. '..\..\lib' om du lagt lib biblioteket utanför lösningens bibliotek.
- I Project->Properties: Linker->Input
  Lägg in PCLibs20xx.lib (alt. MazeLib20xx.lib) i "Additional Dependencies".
- I Project->Properties: C/C++->Advanced
  Ändra "Compile As" till "Compile as C Code (/TC)".

Notera att implementationen av grafikbiblioteket i MazeLib har karaktären av ett ful-hack och därför kan ha en hel del knepigheter för sig. Program som använder detta körs lämpligast inifrån Visual Studio eller från en kommandoprompt (cmd.exe).

Kom ihåg att välja "Manifest Tool - Embed Manifest - No" och "C/C++ - Runtime library - Multithread Debug" för ert projekt om det inte fungerar direkt.

Problem 1: Kapitel 5, uppgift 8, s. 227

En **Graykod** är en binär kodning av heltal med den speciella egenskapen att två efter varandra följande värden alltid skiljer sig i exakt en bit. Denna egenskap gör att Graykoden lämpar sig väl för olika typer av mekaniska eller optiska givare som omvandlar en position till ett digitalt värde på grund av att man inte riskerar att få fullständigt felaktiga värden i övergången mellan lägen.

En tre bitars Graykod (av typen ”binary-reflected”) kodar talen 0 till 7 på följande vis:

1	000
2	001
3	011
4	010
5	110
6	111
7	101
8	100

Notera att varje tal bara skiljer sig från föregående och nästa tal i en bitposition.
En ”binary-reflected” Graykod för *N* bitar är rekursivt uppbyggd och kan skapas enligt följande:

1. Skriv ner Graykoden för *N-1* bitar med en **0**a framför.
2. Skriv ner Graykoden för *N-1* bitar *i omvänd ordning* med en **1**a framför.

I den här uppgiften skall ni skriva en funktion

```
void PrintGrayCode(int nBits)
```

som skriver ut alla värdena i nummerordning för en Graykod med nBits bitar med hjälp av rekursion. Det kan vara lämpligt att använda en (eller två) rekursiv(a) hjälpfunktion(er). Globala variabler *får inte* användas.

Problem 2: Kapitel 5, uppgift 10, s. 229

> I am the only child of parents who weighted, measured, and priced everything; for whom what could not be weighted, measured, and priced had no existense.
>
> —Charles Dickens, Little Dorrit, 1857

In Dicken's time, merchants measured many commodities using weights and a two-pan balance—a practice that continues in many parts of the world today. If you are using a limited set of weights, however, you can only measure certain quantities accurately.

For example, suppose that you have only two weights: a 1-ounce weight and a 3-ounce weight. With these you can easily measure out 4 ounces, as shown:

It is somewhat more interesting to discover that you can also measure out 2 ounces by shifting the 1-ounce weight to the other side, as follows:

Write a recursive function

```
bool IsMeasurable(int target, int weights[], int nWeights)
```

that determines whether it is possible to measure out the desired target amount with a given set of weights. The available weights are stored in the array **weights**, which has **nWeights** as its effective size. For instance, the sample set of two weights illustrated above could be represented using the following pair of variables:

```
static int sampleWeights[] = { 1, 3 };
   static int nSampleWeights = 2;
```

Given these values, the function call

```
IsMeasurable(2, sampleWeights, nSampleWeights)
```

should return **TRUE** because it is possible to measure out 2 ounces using the sample weight set as illustrated in the preceding diagram. On the other hand, calling

```
IsMeasurable(5, sampleWeights, nSampleWeights)
```

should return **FALSE** because it is impossible to use the 1- and 3-ounce weights to add up to 5 ounces.

The fundamental observation you need to make for this problem is that you can use each weight in any of the following three ways:

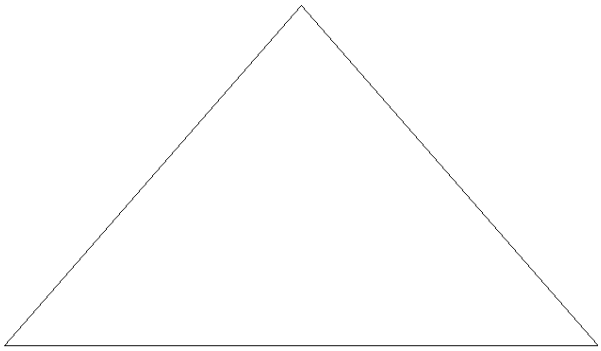
1. You can put it on the opposite side of the balance from the sample.
2. You can put it on the same side of the balance as the sample.
3. You can leave it off the balance entirely.

If you consider one of the weights in the array and determine how choosing one of these three options affects the rest of the problem, you should be able to come up with the recursive insight you need to solve the problem.

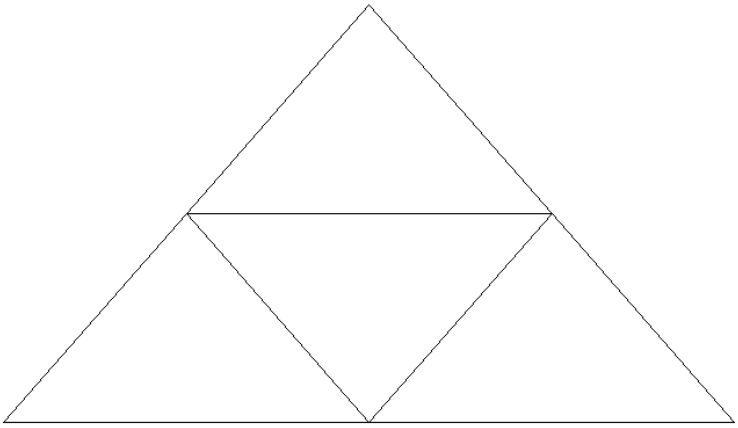
I denna uppgift skall programmet fungera på så sätt att användaren skall mata in en uppsättning vikter varefter programmet skall skriva ut vad som är möjligt att väga upp med de givna vikterna. På samma sätt som i uppgift 1 är det bara själva nyckelfunktionen som behöver vara rekursiv. Observera att det aldrig kan vara fler än tio vikter varför ni kan deklarera vektorn statiskt. Även här får globala variabler inte förekomma. Notera också att en funktion som enligt sin prototyp returnerar ett värde av en viss typ ovillkorligen måste göra det.

Problem 3: Rekursiv uppdelning av triangel

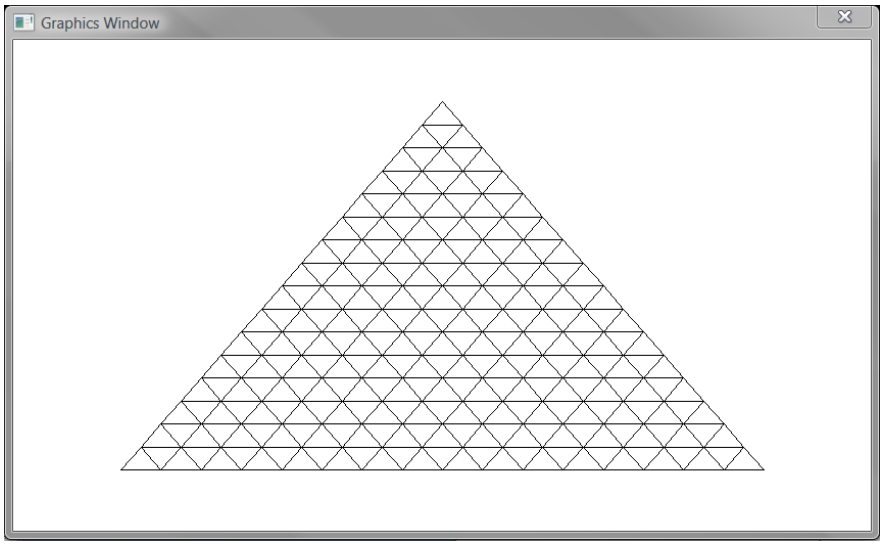
En triangel bestäms av sina tre hörn, så om man har positionen för tre punkter kan man rita en triangel.



Om man delar triangelns sidor på mitten och förbinder dessa med linjer fås 4 mindre trianglar.



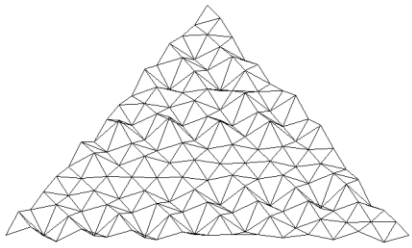
Var och en av dessa kan i sin tur delas in i fyra nya trianglar på samma vis. Skapa ett program som med hjälp av en rekursiv funktion ritar en triangel uppdelad ett godtyckligt antal gånger enligt ovan. I denna uppgift skall användaren inte mata in något utan ritandet börjar så fort programmet startas. Observera att ni måste inkludera graphics.h.



Extrauppgift:

Om man lägger till en oregelbunden förskjutning av varje ny punkt så fås en viss 3-dimensionell effekt. Förskjutningen måste dock vara helt bestämd av punktens läge eftersom det annars kan bli glipor mellan näraliggande trianglar (varför?). I bilden nedan är förskjutningen bestämd av

```
p.y += sin(10.0*p.x + 5.0 * p.y) * 0.15 * pow(0.5, level).
```



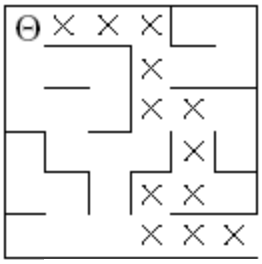
Problem 4: Kapitel 6 övningsuppgift 2, 3 och 4 s. 274

Följande tre uppgifter ger dig möjlighet att studera några intressanta aspekter med backtracking. Vi tillhandahåller ett bibliotek MazeLib20xx.lib som implementerar gränssnittet mazelib.h från Figur 6-1 i kursboken på sidorna 240-241. Vi tillhandahåller även koden för funktionen SolveMaze i filen maze.c som du kan använda som utgångspunkt.

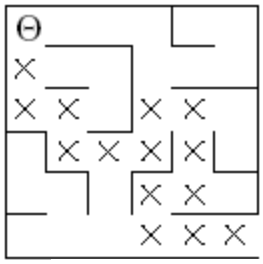
Labyrinten som används i avsnitt 6-1 i boken finns i filen example.maz. Innan du börjar lösa uppgifterna nedan är det lämpligt att du skapar ett projekt och testkör maze.c för att se hur det fungerar.

a) Övningsuppgift 2 på sidan 274.

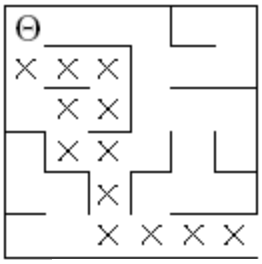
In many mazes, there are multiple paths. For example, the diagrams below show three solutions for the same maze:



length = 13

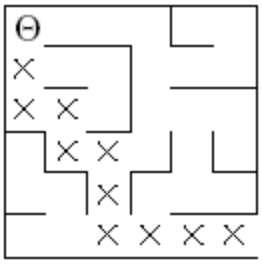


length = 15



length = 13

None of these solutions, however, is optimal. The shortest path through the maze has a path length of 11:



Write a function

```
int ShortestPathLength(pointT pt);
```

that returns the length of the shortest path in the maze from the specified position to any exit. If there is no solution to the maze, **ShortestPathLength** should return the constant **NoSolution**, which is defined to have a value larger than the maximum premissible path length, as follows:

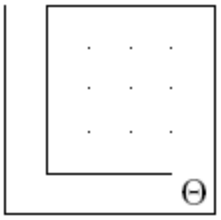
```
#define NoSolution 10000
```

Labyrinten ovan finns i filen pathlen.maz. Observera att användaren skall ange filnamnet på den labyrint som skall testas. Programmet skall utgå från att den filen ligger i samma katalog som den körbara filen.

b) Övningsuppgift 3 på sidan 275. Den här uppgiften går ut på att modifiera funktionen **`SolveMaze`** så att den returnerar ytterligare information till anropande funktion, nämligen antalet rekursiva anrop som utförts. Då returvärdet från funktionen indikerar om labyrinten har en lösning eller inte så är det enklast att använda någon annan mekanism för att returnera antalet rekursiva anrop som utförts. Detta är ett av de få tillfällen då man kan tänka sig att använda en global variabel, även om det inte är nödvändigt.

As implemented in Figure 6-3, the **`SolveMaze`** function unmarks each square as it discovers there are no solutions from that point. Although this design strategy has the advantage that the final configuration of the maze shows the solution path as a series of marked squares, the decision to unmark squares as you backtrack has a considerable cost in terms of the overall efficiency of the algorithm. If you've marked a square and then backtracked through it, you've already explored the possibilities leading from that square. If you come back to it by some other path, you might as well rely on your earlier analysis instead of exploring the same option again.

To give yourself a sense of how much these unmarking operations cost in terms of efficiency, extend the **`SolveMaze`** program so that it records the number of recursive calls as it proceeds. Use this program to calculate how many recursive calls are required to solve the following maze if the call to **`UnMarkSquare`** remains part of the program:



Run your program again, this time without the call to **`UnmarkSquare`**. What happens to the number of recursive calls?

Labyrinten ovan finns i filen **`unmark.maz`**. Ni skall i denna uppgift lämna in en körbar fil där två tester görs; enda skillnaden är huruvida anropet till UnmarkSquare är kvar eller inte. Användaren skall på samma sätt som i uppgift a) mata in filnamnet på labyrinten. Var vänliga att skruva upp hastigheten på den lösning ni lämnar in.

c) Övningsuppgift 4 på sidan 275. I den här uppgiften skall du skriva ytterligare en modifikation av funktionen **`SolveMaze`**. Hittills har vi sett att avmarkeringen av rutor under backtrackingen låter oss se vägen till utgången då algoritmen terminerar (uppgift a). Vi har dock konstaterat att själva avmarkeringen av rutor har en oönskad effekt på algoritmens effektivitet (uppgift b). I den här uppgiften kommer vi inte att avmarkera rutorna utan istället hålla reda på vägen explicit i ett fält så att vi vägen finns sparad då algoritmen terminerar. Notera att det sista argumentet till **`FindPath`**(**`MaxPath`**) i den här uppgiften är fältets allokerade storlek. Tips! Det är lämpligt att skriva en hjälpfunktion som håller reda på längden på den aktuella vägen. Notera att globala variabler inte behövs eller bör användas i den här deluppgiften.

As the result of the preceding exercise makes clear, the idea of keeping track of the path through a maze by using the **MarkSquare** facility in **mazelib.h** has a substantial cost. A more practical approach is to change the definition of the recursive function so that it keeps track of the current path as it goes. Following the logic of **SolveMaze**, write a function

```
int FindPath(pointT pt, pointT path[], int maxPathSize);
```

that takes, in addition to the coordinates of the starting position, an array of **pointT** values called **path** whose allocated size is **maxPathSize**. When **FindPath** returns to the calling program, the elements of the **path** array should be initialized to a sequence of coordinates beginning with the starting position and ending with the coordinates to the first square that lies outside the maze. The function returns the number of points in the actual path, or 0 if no path exists or if the path array would overflow. For example, if you use **FindPath** with the following main program, it will display the coordinates of the points in the solution path on the screen.

```
main()
{
    pointT path[MaxPath];
    int i, len;

    ReadMazeMap(MazeFile);
    len = FindPath(GetStartPosition(), path, MaxPath);
    if (len == 0) {
        printf("No solution exists.\n");
    } else {
        printf("The following path is a solution:\n");
        for (i = 0; i < len; i++) {
            printf(" (%d, %d)\n", path[i].x, path[i].y);
        }
    }
}
```

For this exercise, it is sufficient for **FindPath** to find any solution path. It need not find the shortest one.

Att lämna in

På denna laboration skall ni lämna in elektroniskt via PingPong. Skapa ett arkiv (zip, rar eller liknande) med era lösningsprojekt och skicka med det arkivet när ni lämnar in i PingPong. Det skall finnas ett projekt för varje deluppgift (alltsammans kan vara i ett Visual Studio-projekt). T.ex.

GrayCode	-	Problem 1
Weights	-	Problem 2
DrawTriangle	-	Problem 3
ShortestPath	-	Problem 4, uppgift a)
Unmark	-	Problem 4, uppgift b)
FindPath	-	Problem 4, uppgift c)

Ni kan se önskad funktionalitet genom att jämföra med exempellösningarna

Bedömning

En fungerande lösning med korrekt kod ger på denna laboration godkänt - naturligtvis under förutsättning att instruktionerna följts. Endast om koden är klart undermålig i något avseende kommer den att underkännas.

Lycka Till!