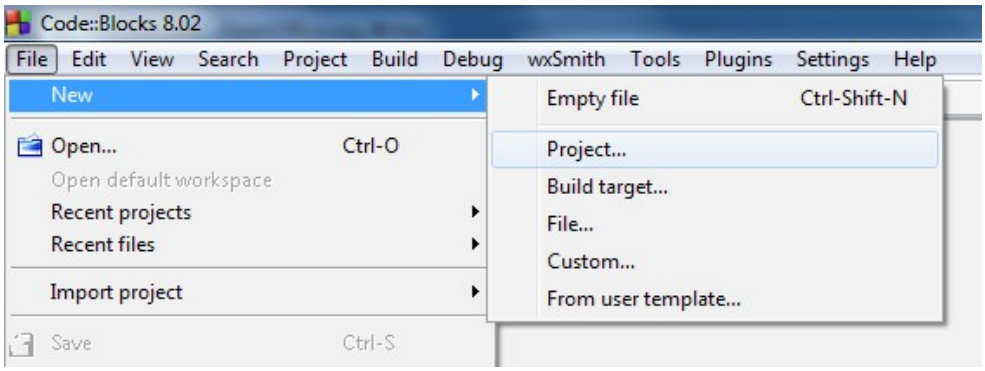
Föreläsning 5 - SDL

OBS: Inför denna föreläsning är det viktigt att ni testat att installera SDL, följ alltså installationsanvisningarna i sin helhet om ni inte redan gjort det.

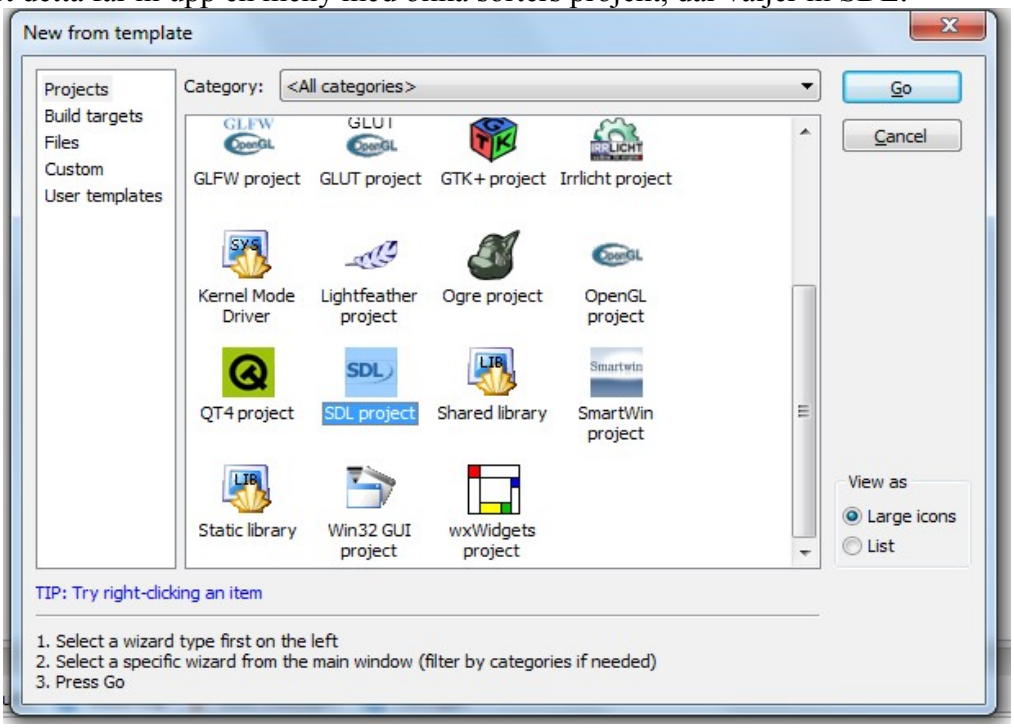
Vi ska nu förhoppningsvis ha lite roligare än tidigare, vi ska börja med grafik. Grafik har två fördelar,

1. Vi kan se på skärmen vad våra program utför
2. Det är lite roligare att rita saker istället för att bara köra konsolprogram (svarta rutan)

Det förutsätts som sagt att ni installerat och provkört SDL inför denna föreläsning. Då man använder SDL i Code Blocks gör man så här, först väljer man New-Project under filmenyn:

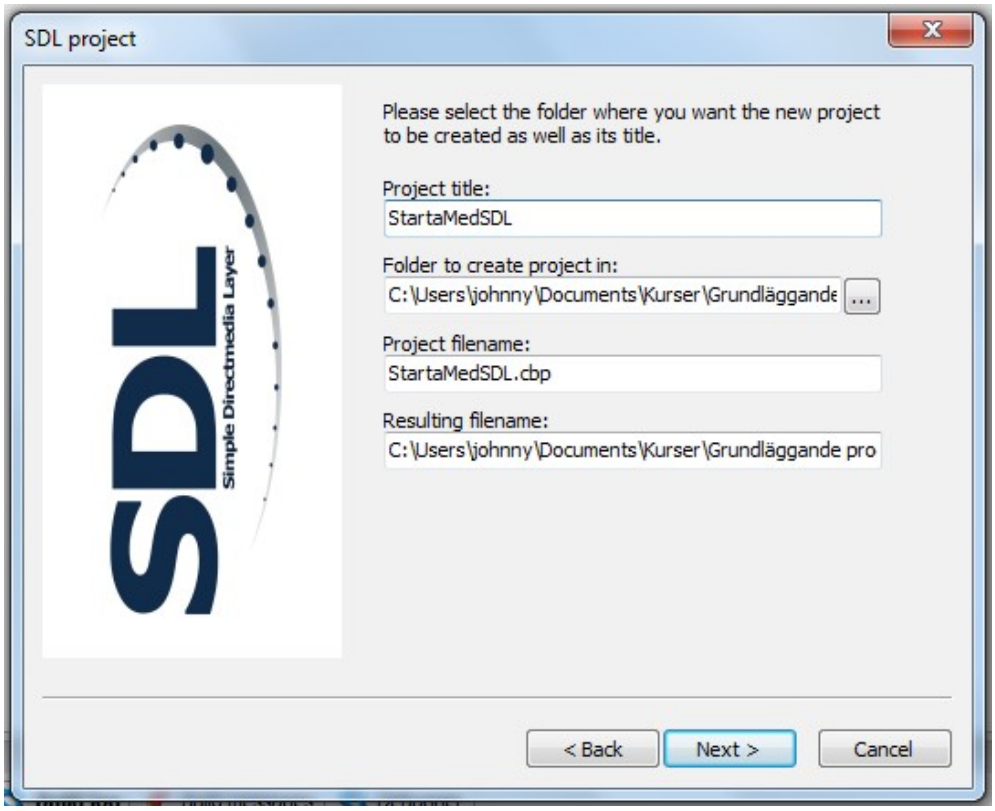


När ni gjort detta får ni upp en meny med olika sorters projekt, där väljer ni SDL:



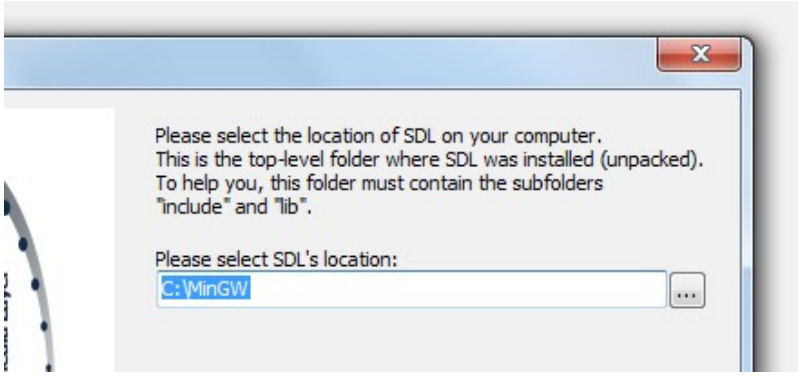
Sedan kommer en till sån där Wizard där man får välja projektnamn och plats, det spelar inte så stor roll nu i början, jag valde så här:

Johnny Panrike, johnny.panrike@sth.kth.se



Ni kan göra något liknande. På sikt måste ni förstås skapa er en egen filorganisation som stödjer er när ni får många program och laborationer att hålla reda på, men det får var och en ordna själv efter egen smak.

I dialogruta indikerar man var SDL är installerat,

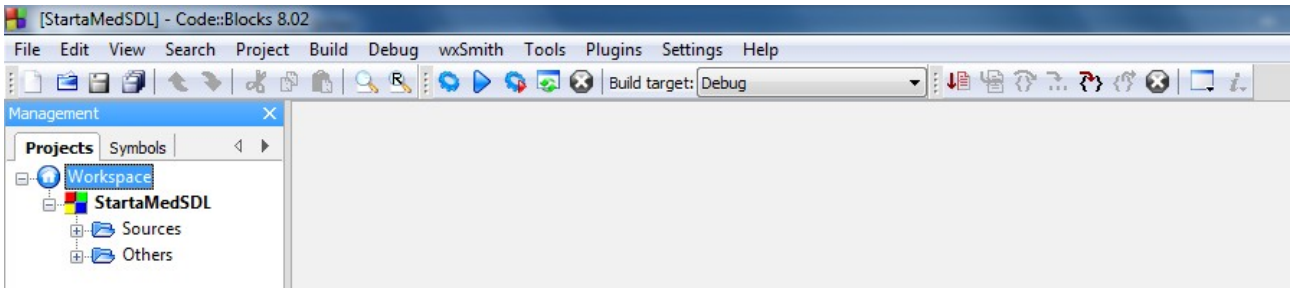


vi har redan pratat en del om det i installationsanvisningarna så vi nämner ingenting om det här. Det bör fungera, annars får man se på sina installationer.

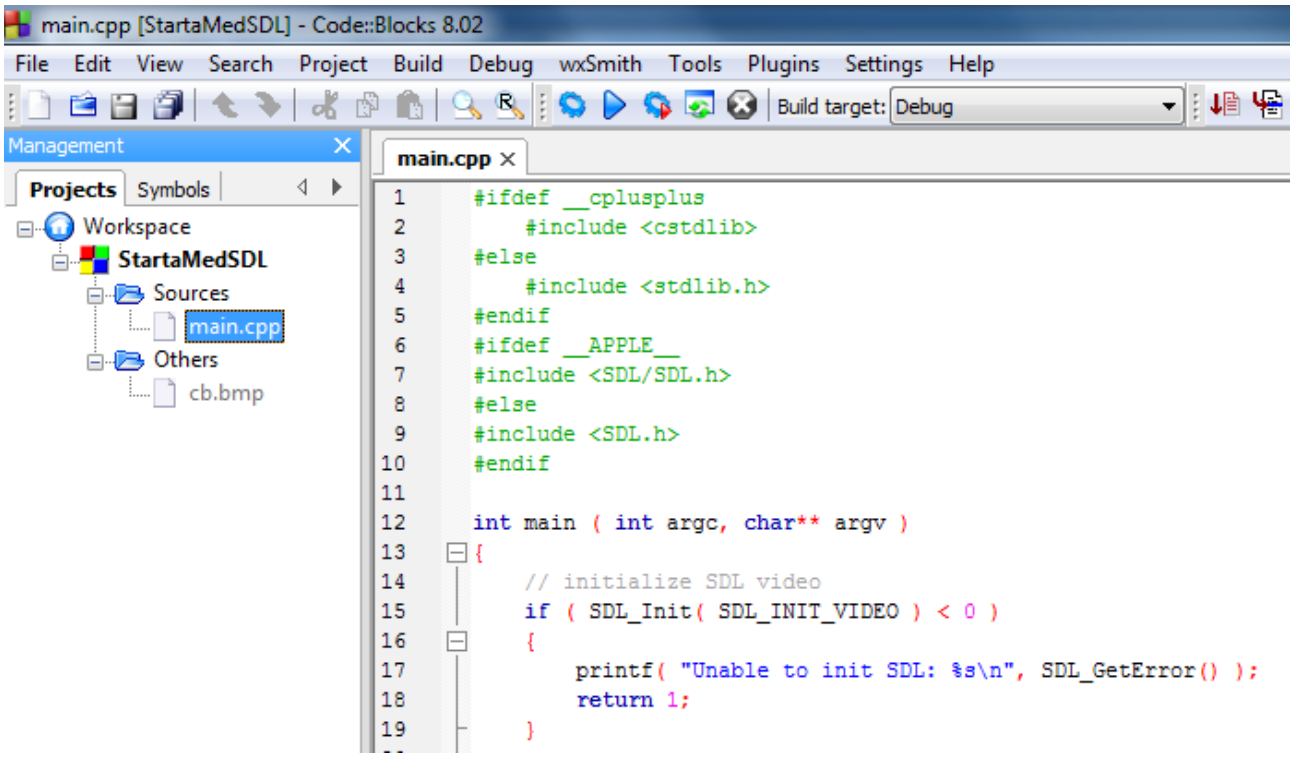
I nästa dialog ska man indikera vilken kompilator man vill använda, samt ange en del katalogval, troligtvis är det även här bara att klicka sig vidare. Detta är sista rutan så här klickar man sig vidare genom ”Finish” och då får man förhoppningsvis igång grafikprogrammeringen.

Johnny Panrike, johnny.panrike@sth.kth.se

Vi har nu en vy i stil med följande:

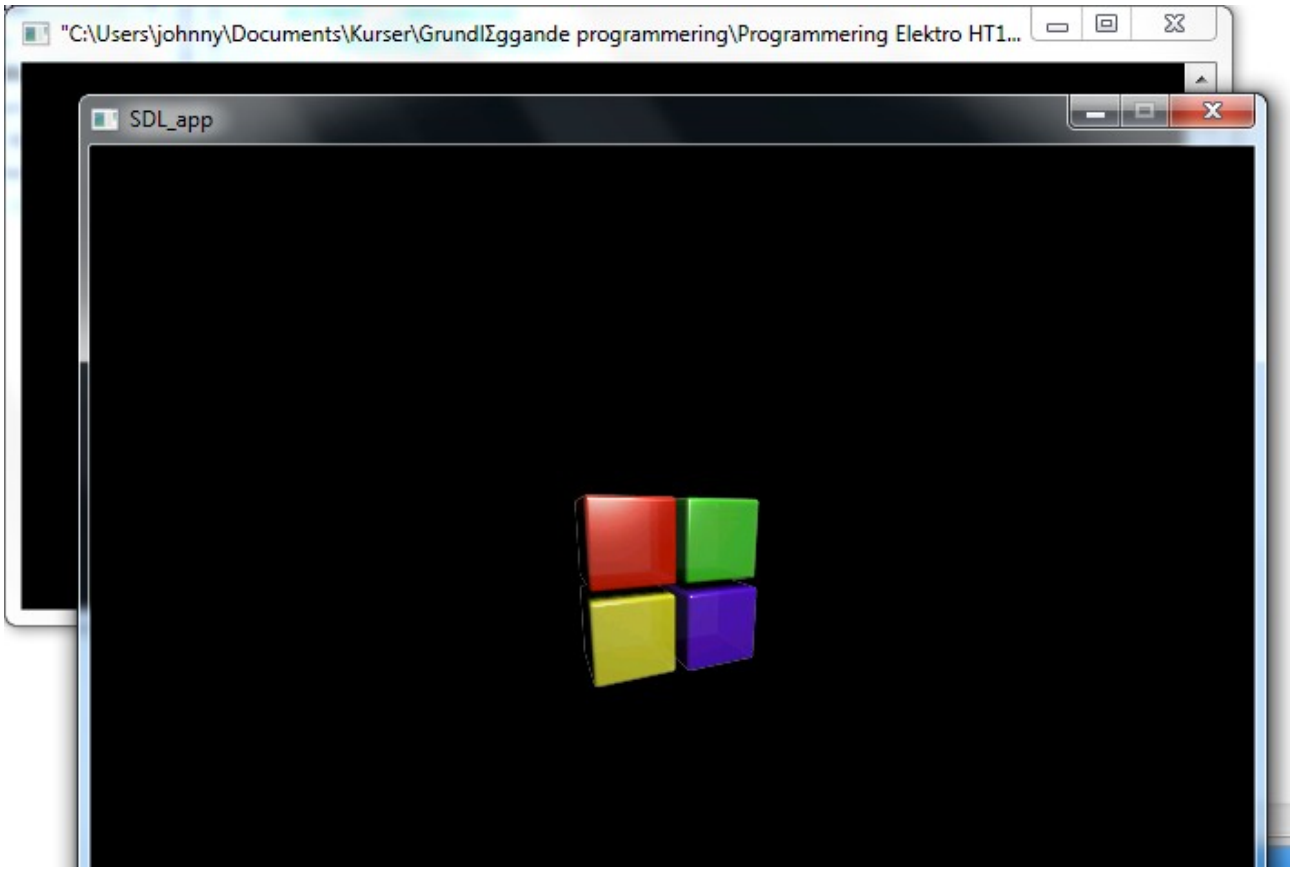


Om vi förstorar vänstra delen av bilden och öppnar Sources och Others genom att klicka på plustecknena får vi följande vy:



Vi har här även klickat på ”`main.cpp`” som innehåller ett program som är själva starten för det grafikprogram som vi ska utvidga så att det blir som vi vill ha det. När vi gör grafikprogrammering på det här sättet behöver vi alltså inte börja från absoluta början, vi får en liten skjuts alltså. Dock ska vi vara medvetna om att det här programmet som är vår startpunkt inte är ett vanligt *C*-program, som vi varit vana vid hittills, det är faktiskt ett *C*++-program. Vi ser det genom att den har ändelsen `.cpp` istället för `.c`. Lyckligtvis är språket *C*, som är temat för denna kurs, i stort sett en del av språket *C*++ så allt som fungerar i *C* fungerar också i *C*++. Ni kan alltså lugnt fortsätta programmera i *C* även fast kompilatorn tolkar det ni skriver som *C*++.

Vi ser i programmet att det står ”`main`” och det är samma main som vanligt även om vi har en del mer krafs runtikring. Detta program kan köras genom att vi (som vanligt) trycker ctrl-F9, ctrl-F10. Vi får då följande vy:



Det är, som syns, *två* fönster, ett i bakgrunden som det är det gamla vanliga konsolfönstret som vi kan mata ut text genom. Framför det så visas grafikfönstret. Grafikfönstret kan INTE visa text, bara grafik, vi kan alltså inte mata in text i det. Vi ser en bild som är själva Code Blocks logotype som visas i grafikfönstret. Vi kan stänga grafikfönstret genom att klicka på den röda stängknappen. Då är inte programmet avslutat ännu, konsolfönstret ligger kvar och vi måste stänga även det för att kunna komma tillbaka till Code Blocks för att programmera.

Vi ska ni se på hur vi själva ska kunna komma igång och programmera grafiskt. Vi ska börja med att sätta olika färger på enskilda bildelement, så kallade pixels, i det grafiska fönstret. Men vi börja först med att rensa upp lite i startprogrammet, vi vill ju inte alltid rita ut Code Blocks logotype, eller hur?

Startprogrammet visas nedan. Vi ser att det är *C*, men också att det är flera saker som vi inte gått igenom ännu. Vi ska rensa i det här programmet för att dels göra det lättare att läsa och dels för att förbereda det för kommande grafikprogrammering. Några rader nedan är i stor, fet stil, hitta dem!

```
#ifdef __cplusplus
    #include <cstdlib>
#else
    #include <stdlib.h>
#endif
#ifdef __APPLE__
#include <SDL/SDL.h>
#else
#include <SDL.h>
#endif
```

Johnny Panrike, johnny.panrike@sth.kth.se

```
int main ( int argc, char** argv )
{
    // initialize SDL video
    if ( SDL_Init( SDL_INIT_VIDEO ) < 0 )
    {
        printf( "Unable to init SDL: %s\n", SDL_GetError() );
        return 1;
    }

    // make sure SDL cleans up before exit
    atexit(SDL_Quit);

    // create a new window
    SDL_Surface* screen = SDL_SetVideoMode(640, 480, 16,
                                           SDL_HWSURFACE|SDL_DOUBLEBUF);
    if ( !screen )
    {
        printf("Unable to set 640x480 video: %s\n", SDL_GetError());
        return 1;
    }

    // load an image
    SDL_Surface* bmp = SDL_LoadBMP("cb.bmp");
    if (!bmp)
    {
        printf("Unable to load bitmap: %s\n", SDL_GetError());
        return 1;
    }

    // centre the bitmap on screen
    SDL_Rect dstrect;
    dstrect.x = (screen->w - bmp->w) / 2;
    dstrect.y = (screen->h - bmp->h) / 2;

    // program main loop
    bool done = false;
    while (!done)
    {
        // message processing loop
        SDL_Event event;
        while (SDL_PollEvent(&event))
        {
            // check for messages
            switch (event.type)
            {
                // exit if the window is closed
            case SDL_QUIT:
                done = true;
                break;

                // check for keypresses
            case SDL_KEYDOWN:
                {
                    // exit if ESCAPE is pressed
                    if (event.key.keysym.sym == SDLK_ESCAPE)
                        done = true;
                    break;
                }
```

```
            } // end switch
        } // end of message processing

        // DRAWING STARTS HERE

        // clear screen
        SDL_FillRect(screen, 0, SDL_MapRGB(screen->format, 0,0, 0));

        // draw bitmap
        SDL_BlitSurface(bmp, 0, screen, &dstrect);

        // DRAWING ENDS HERE

        // finally, update the screen :)
        SDL_Flip(screen);
    } // end main loop

    // free loaded bitmap
    SDL_FreeSurface(bmp);

    // all is well ;)
    printf("Exited cleanly\n");
    return 0;
}
```

Vi tar ut de speciella raderna och förklara dem rad för rad (vi behåller indenteringen):

```
int main ( int argc, char** argv )
```

Detta är den vanliga början på ett *C*-program, visst vi ser lite extra detaljer, men de behöver vi inte fästa något avseende vid just nu. Innan finns en del inkluderingsdirektiv som även har en del villkorssatser i sig, vi kan bara låta dem stå som de är och inkludera extra saker om vi behöver det framöver. Vi kommer absolut att behöva `math.h` så då kommer vi att få utseendet

```
#ifdef __cplusplus
    #include <cstdlib>
#else
    #include <stdlib.h>
#endif
#ifdef __APPLE__
#include <SDL/SDL.h>
#else
#include <SDL.h>
#endif

#include <math.h>
```

då vi utökar listan på bibliotek vi behöver. Observera att `stdio.h` inte är med här, detta är bara ett grafikprogram som inte hanterar in- och utmatning via `stdio.h`.

En bit ned i programmet ser vi raderna

```
// program main loop
bool done = false;
```

När vi kommmit hit har vi genomför alla initieringar och är redo att starta den så kallade ”event-loopen”. Det är en loop som kör ända tills användaren klickar på röda stängrutan eller trycker på escape. Då sätts variabeln `done` till `true`. Datatypen `bool` finns inte i *C* utan kommer från *C++* och är en typ som endast kan ha två värden, `true` respektive `false`. Man kan uppnå samma sak i *C* genom deklarationen

```
int done = 0;
```

och anse att `0` betyder falskt och sedan när man vill att done ska innehåla värdet sant så sätter man `done = 1;`. Varför krångla med `true` och `false` då, varför inte bara sätta 0 och 1? Ja, det är en attitydfråga, det visar sig vara bra att använda värdena `true` och `false` och *C++* är ett mycket bättre språk än *C*.

Men vi behöver inte dröja mer vid det här, det intressanta här är att det finns en stor loop som kör som kallas event-loopen. I senare kurser ska ni lära er att hämta upp händelser och styra ett programs beteende via en event-loop, men inte just nu. Det enda vi ska göra i denna kurs är att rita grafik och det räcker att gå ett varv i event-loopen för att göra det. Lite längre ner ser vi hur själva grafiken ritas ut, så här ser det ut:

```
// DRAWING STARTS HERE

// clear screen
SDL_FillRect(screen, 0, SDL_MapRGB(screen->format, 0,0, 0));

// draw bitmap
SDL_BlitSurface(bmp, 0, screen, &dstrect);

// DRAWING ENDS HERE

SDL_Flip(screen);
```

Detta är förstås väldigt väl kommenterat, kommentaren ”**// DRAWING STARTS HERE**” säger förstås mycket. Det är mellan raderna där det står ”**// DRAWING STARTS HERE**” respektive ”**// DRAWING ENDS HERE**” som vi ska lägga in våra grafikkommandon. Nu finns det redan en del kommandon där för att rita ut den logotype vi såg tidigare, vi ska ta bort dem kommandona och även ta bort en del initeringar som hör till dessa kommandon. Vi kommer att göra det inom kort, vi nämner bara några ord om den sista fetmarkerade raden:

```
SDL_FreeSurface(bmp);
```

Denna hör till det som ska bort, logotypen för Code Blocks sparades i en så kallad bitmap och den lagras i variabeln bmp. Innan man avslutar programmet så måste man göra **`SDL_FreeSurface()`** på de bitbilder (bitmappar) man använt.

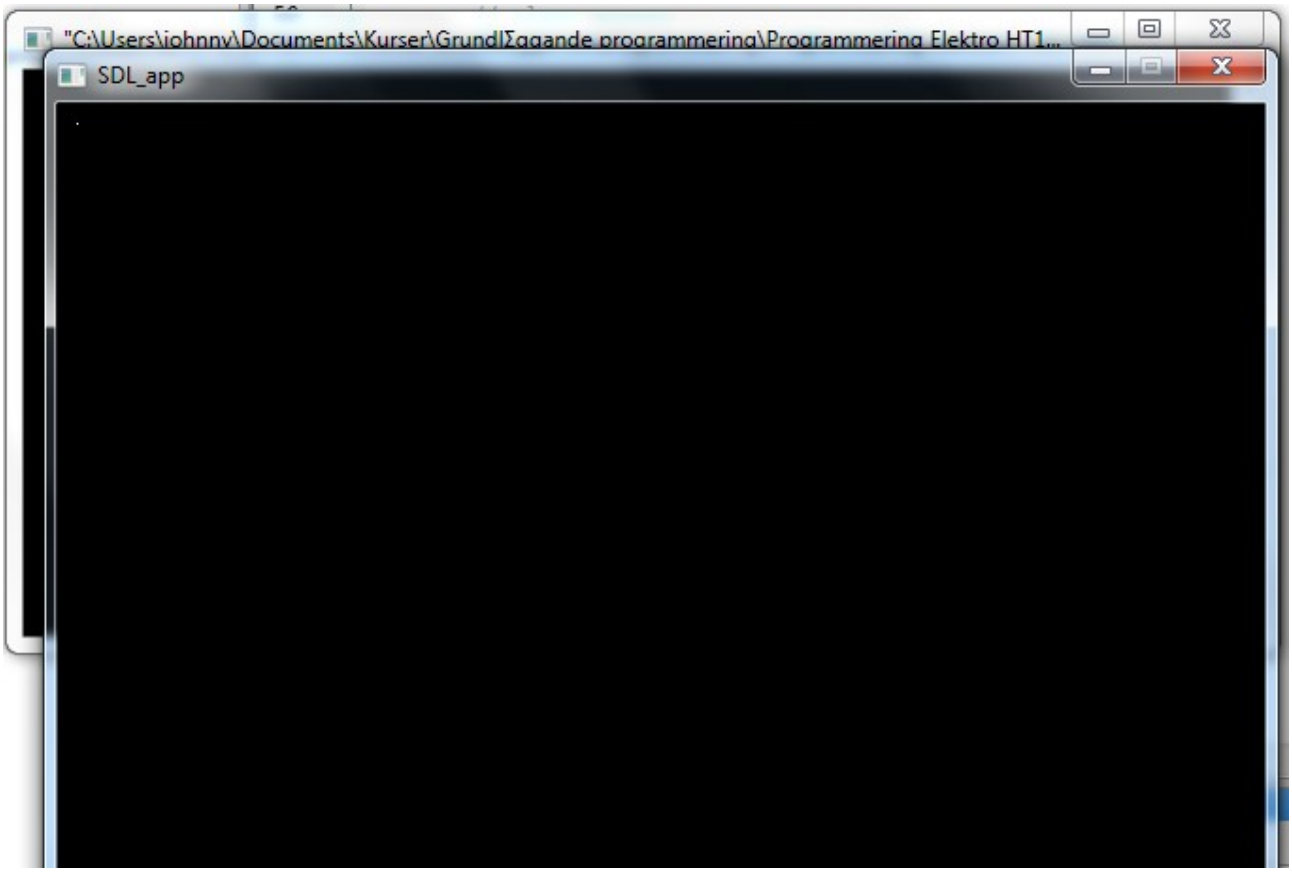
Vi tar nu bort allt som vi inte behöver. Programmet ser då ut så här:

```
#ifdef __cplusplus
...
#endif

int main ( int argc, char** argv )
{
    // initialize SDL video
    if ( SDL_Init( SDL_INIT_VIDEO ) < 0 )
    {   printf( "Unable to init SDL: %s\n", SDL_GetError() );
        return 1; }

    // make sure SDL cleans up before exit
    atexit(SDL_Quit);

    // create a new window
    SDL_Surface* screen = SDL_SetVideoMode(640, 480, 16,
                                           SDL_HWSURFACE|SDL_DOUBLEBUF);
    if ( !screen )
    {   printf("Unable to set 640x480 video: %s\n", SDL_GetError());
        return 1; }

    // program main loop
    bool done = false;
    while (!done)
    {   // message processing loop
        SDL_Event event;
        while (SDL_PollEvent(&event))
        {   // check for messages
            switch (event.type)
            {
            case SDL_QUIT: // exit if the window is closed
                done = true;
                break;
            case SDL_KEYDOWN: // check for keypresses
                {   // exit if ESCAPE is pressed
                    if (event.key.keysym.sym == SDLK_ESCAPE)done = true;
                    break;
                }
            } // end switch
        } // end of message processing

        // DRAWING STARTS HERE

        // clear screen
        SDL_FillRect(screen, 0, SDL_MapRGB(screen->format, 0, 0, 0));

        //Här kan vi lägga in våra grafikkommandon!

        // DRAWING ENDS HERE

        SDL_Flip(screen); // finally, update the screen :)
    } // end main loop
    // all is well ;)
    printf("Exited cleanly\n");
    return 0;
}
```

Jag har valt att frångå indenteringsdisciplinen här för att få in programmet på en sida, vidare har jag också utelämnat alla förprocessordirektiv (de där sakerna med # först) ovan, men alla ska finnas med. Detta är nu ett startprogram som vi kan använda för att rita på skärmen. Där det står ”`//Här kan vi lägga in våra grafikkommandon!`” kan, ja, alltså lägga in grafikkommandon. MEN: Vi har inte sett några grafikkommandon ännu, förutom utritning av logotypen. Vi introducerar nu därför funktionen `DrawPixel()` som anropas så här:

```
DrawPixel(screen, 255, 255, 255, 10, 10);
```

I anropet så skriver man först den yta som man vill rita på, i vårt program så finns en variabel som heter screen som refererar till ritytan i det grafiska fönstret. Det efterföljande tre heltalen är tal mellan 0 och 255 som anger RGB-värden (R=Red, G=Green, B=Blue) för den pixel man vill rita, genom att välja olika värden kan vi få (i princip) vilken färg vi vill. Slutligen anger de sista två heltalen x- och y-koordinater (utgående från övre vänstra hörnet) för den punkt som ska påverkas. Ovanstående kommando gör alltså pixeln i 10,10 vit och vi får följande resultat:



och här är kodavsnittet i programmet `cleanedupmainwithdrawpixel.cpp`:

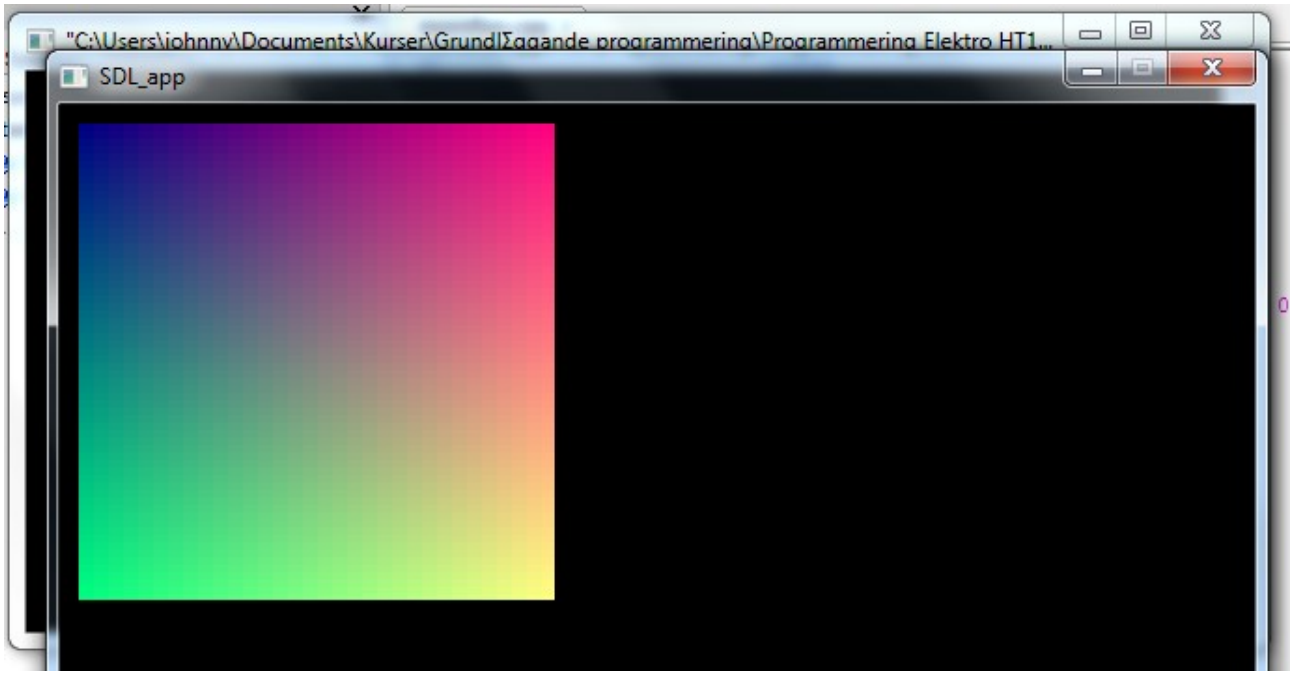
```
// DRAWING STARTS HERE
// clear screen
SDL_FillRect(screen, 0, SDL_MapRGB(screen->format, 0, 0, 0));
//Här kan vi lägga in våra grafikkommandon!
DrawPixel(screen, 255, 255, 255, 10, 10);
// DRAWING ENDS HERE
SDL_Flip(screen); // finally, update the screen :)
```

Johnny Panrike, johnny.panrike@sth.kth.se

Inte så upphetsande, kanske det kan tyckas, men låt oss göra något mer intressant. Vi gör en loop-konstruktion som låter oss rita mer pixlar och vi grejar lite med färger också, vi lägger in koden:

```
// DRAWING STARTS HERE

// clear screen
SDL_FillRect(screen, 0, SDL_MapRGB(screen->format, 0, 0, 0));

//Lite mera kul:
for(int x = 0;x<255;x++)
    for(int y=0;y<255;y++)
        DrawPixel(screen, x, y, 127, x+10, y+10);

// DRAWING ENDS HERE
```

och får resultatet:



Fint vad? Rött, blått, grönt och gult. Vad har vi gjort egentligen? Låt oss titta närmare på själva konstruktionen, den såg ut så här:

```
for(int x = 0;x<255;x++)
    for(int y=0;y<255;y++)
        DrawPixel(screen, x, y, 127, x+10, y+10);
```

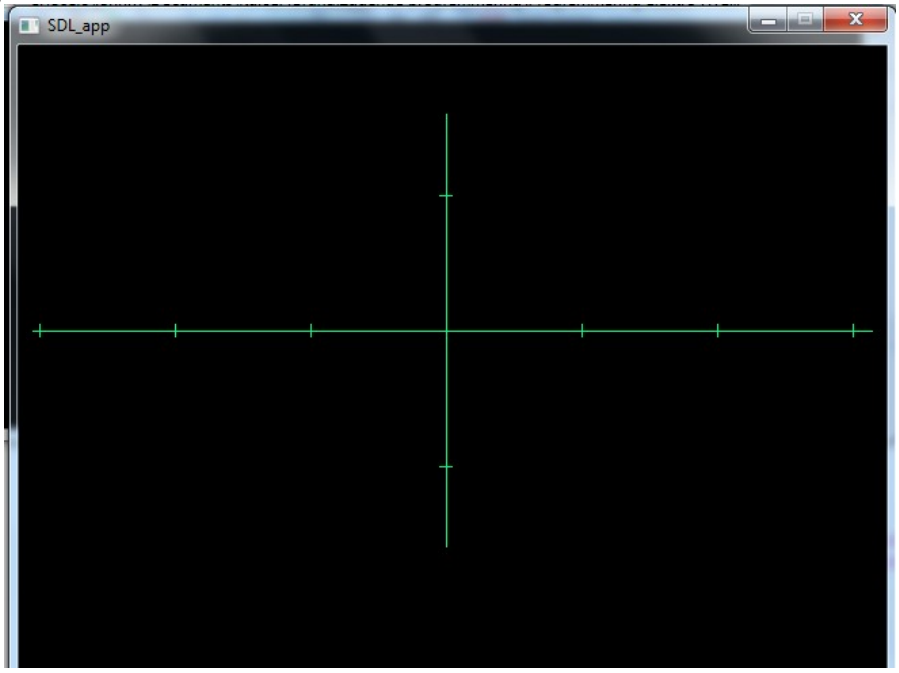
Först en yttre loop med en loopvariabel `x` som går från 0 till 255. Sedan adderar vi en inre loop där `y`, för varje `x`, går från 0 till 255. Vi genererar alltså 65536 par av heltal `x` och `y` (0, 0), (0, 1), (0, 2) och så vidare upp till (255, 253), (255, 254) och (255, 255). För varje par av `x` och `y` går vi till punkten `(x+10, y+10)` på ritytan (`screen`) och målar den pixeln i färgerna som anges av `(x, y, 127)` – för små värden på `x` och `y` blir detta nära `(0, 0, 127)` som är blått och stämmer överens med bilden ovan, små värden på `x` och `y` innebär att vi är nära origo, som är övre vänstra hörnet och den ytan är ju mycket riktigt också blåaktig, eller hur? Origo är alltså här övre vänstra hörnet, inte nedre vänstra hörnet som man kanske skulle vara van vid från då man gör

grafiska framställningar av matematiska funktioner med papper och penna som ni säkert gjort mycket i gymnasiet. När vi arbetar med datorgrafik är koordinatsystem annorlunda i det att just origo ligger högst upp till vänster. Anledningen till detta är förstås att datorgrafik ofta förekommer i dialogrutor (ett finare ord för fönster som man klickar på) som användaren kan påverka storleken av, användaren påverkar storleken av en dialogruta genom att ta tag i en storleksändrare som finns i nedre högra hörnet. Då är det bra att den fasta punkten i en dialogruta ligger överst till vänster vilket alltså är anledningen till att origo (den enda punkt som garanterat finns i ett grafikfönster) ligger fast i övre vänstra hörnet.

Nu ska vi göra ett mer intressant program som ritar upp matematiska funktioner så som vi är vana att se dem, vi ritar först upp ett koordinatsystem och låter origo (alltså som våra funktionerna ska presenteras kring) vara i `(315, 210)`. Vi åstadkommer detta med denna kod:

```
//x-axel:
for(int x=0;x<620;x++)DrawPixel(screen, 0, 255, 127, x+10, 200 + 10);

//Gradbetckningar x-axeln:
for(int x=0;x<4;x++)for(int y=-5;y<5;y++)
{
    DrawPixel(screen,0,255,127,315+x*100,210+y);
    DrawPixel(screen,0,255,127,315-x*100,210+y);
}

//y-axel:
for(int y=0;y<320;y++)DrawPixel(screen, 0, 255, 127, 315, 50+y);

//Gradbetckningar y-axeln:
for(int x=-5;x<5;x++)
{
    DrawPixel(screen,0,255,127,315+x,210-100);
    DrawPixel(screen,0,255,127,315+x,210+100);
}
```

Det ger oss följande resultat:



En yta att rita ut matematiska funktioner på. Vi vill nu alltså att origo är centralt i bilden. Men vi måste också minnas att `DrawPixel()` (och alla andra grafiska funktioner) arbetar med origo i (1,1) och det är i övre vänstra hörnet. Vi måste alltså *omvandla* dels mellan olika skalor och dels med olika förskjutningar om vi vill framställa matematiska funktioner på ett sätt som är överskådligt. I det här fallet har vi 100 pixlar mellan gradangivelserna på axlarna, vi har alltså en skalfaktor på 100 att ta hänsyn till och förskjutningen är 315 pixlar i x-led och 210 pixlar i y-led. Vi ska nu beskriva en allmän metod för att rita ut matematiska funktioner. Vi ska tillämpa det på funktionerna sinus och cosinus.

Vi tänker så här:

1. För varje x-värde längs med x-axeln i grafen ovan ska ett sinusvärde avsättas i höjdled. Det x-värdet ligger, grafiskt sett (på skärmen alltså), mellan värdena 10 och 620. Mitten är 315 och representerar värdet 0. Vi har således det grafiska x-värdet som går från 10 till 620:

   ```
   for(int x=10; x<620; x++)
   ```

2. För varje sådant x måste vi veta vilken punkt på x-xeln som det motsvarar. Vi har skalfaktorn 100 (det var ju 100 pixlar per enhet) och förskjutningen 315 att ta hänsyn till. Värdet på x-axeln blir alltså `(x-315)/100`. Nu måste vi omvandla till flyttal för variabeln `x` är ju ett heltal och för att räkna med sinus måste vi ha flyttal. Vi inför också en till variabel som vi kallas `real_x` som ska hålla det reella x-värdet som svarar mot punkten på x-axeln som vid `x`, vi deklarerar och initierar värdet så här:

   ```
   double real_x = (double)(x-315)/100;
   ```

3. Ovan har vi alltså räknat ut x-värdet (`real_x`) som kan stoppas in i sinusfunktionen, vi inför därför en ny variabel för att hålla sinusvärdet:

   ```
   double sine_x = sin(real_x);
   ```

4. Nu ska vi sätta ut punkten ritytan och vi genomför därför samma omvandling som ovan, med skalfaktor 100 och en förskjutning. Förskjutningen i y-led är dock 210 eftersom y-värdet noll ligger på höjden 210. Vidare får vi också ett minustecken med i formeln för y eftersom i y-led ligger punkter med lägre koordinater högre upp. Sammantaget blir det formeln:

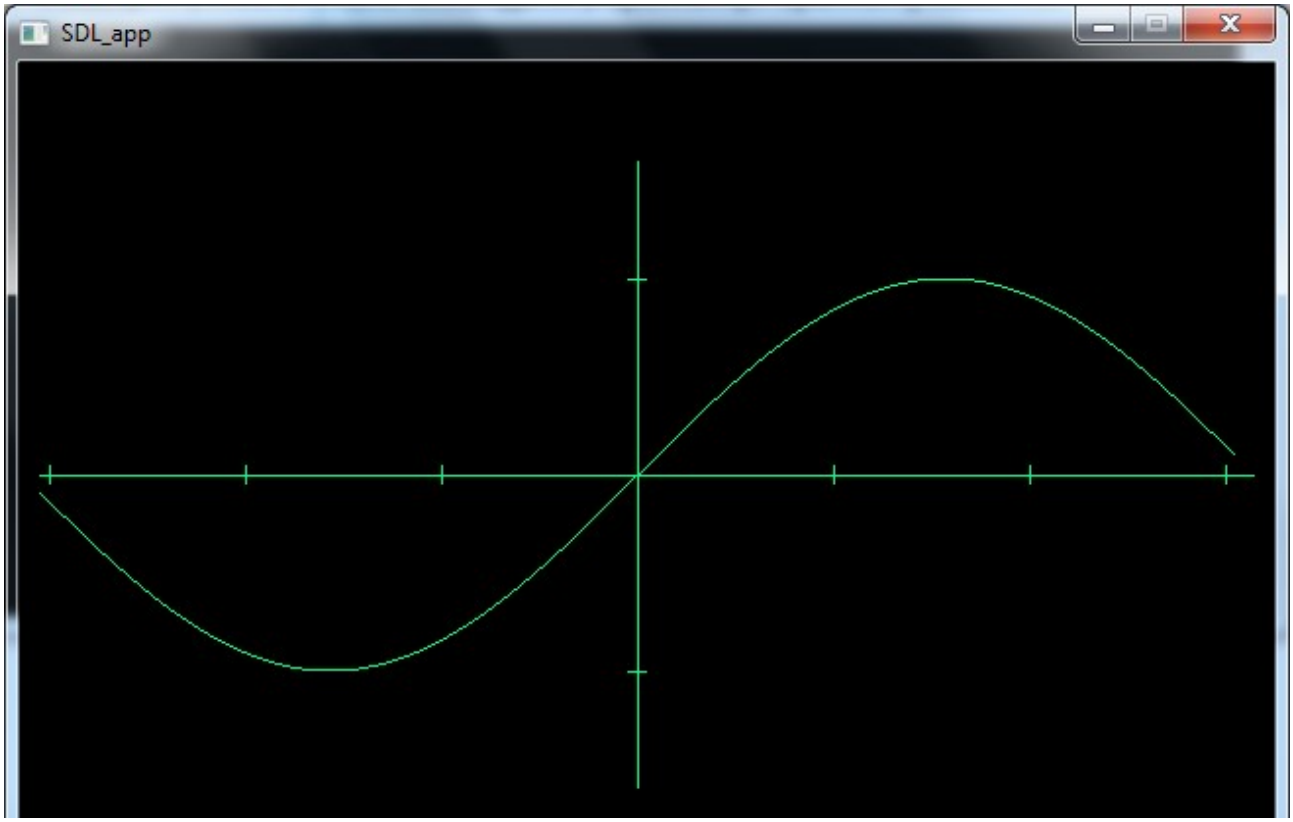
   ```
   int y = 210 - 100*sine_x;
   ```

5. Vi har nu i variabelparet `(x, y)` den punkt som vi vill rita ut på grafen till sinusfunktionen. Vi gör det med `DrawPixel()`:

   ```
   DrawPixel(screen, 0, 255, 127, x, y);
   ```

Hela koden ges på nästa sida tillsammans med resultatet.

```
//Sinus:
for(int x=10; x<620; x++)
{
    double real_x = (double)(x-315)/100;
    double sine_x = sin(real_x);
    int y = 210 - 100*sine_x;
    DrawPixel(screen,0,255,127,x,y);
}
```

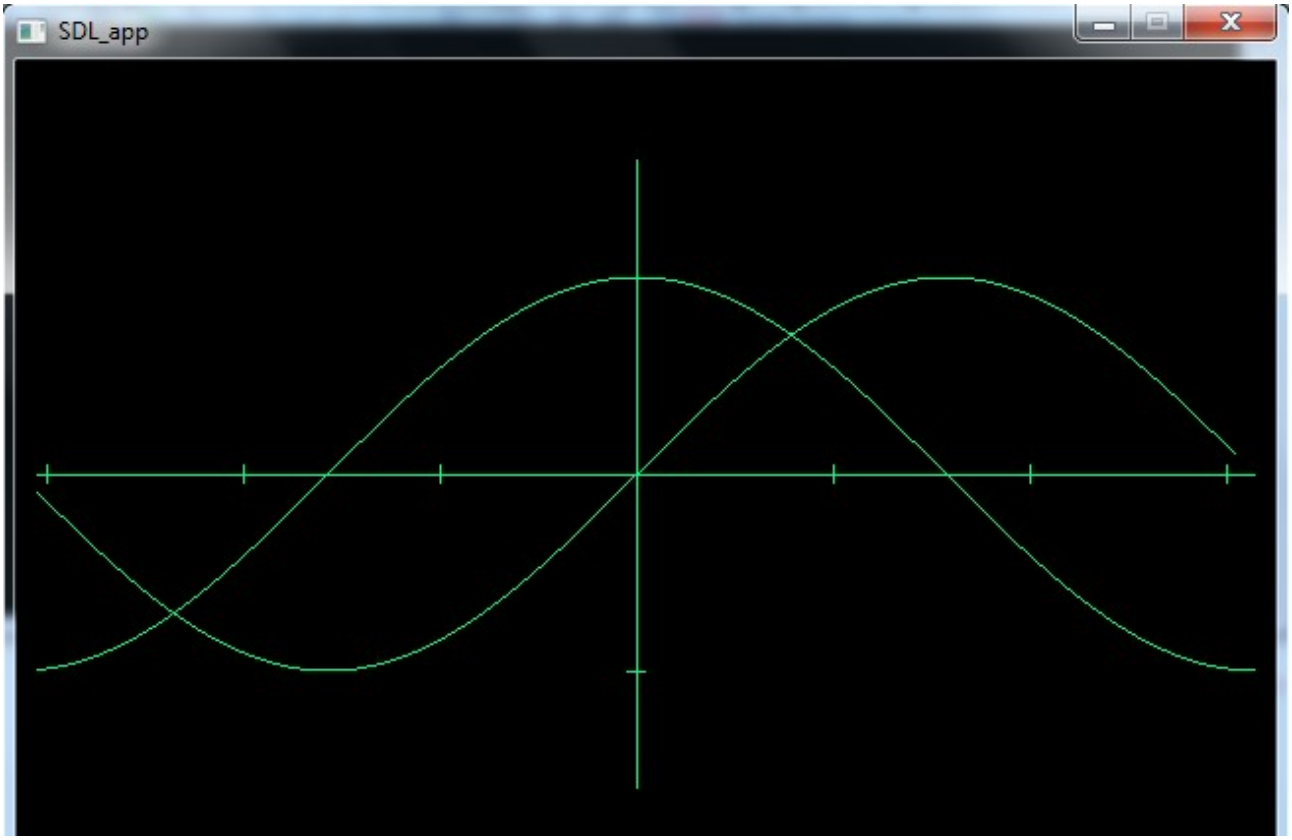


Vi känner nu oss ganska hemma, det är ju så här vi bruka tänka oss grafiska representationer av elementära funktioner. Tänkesättet som är beskrivet i de 5 stegen på föregående sida är förstås helt generellt. Om vi utför samma manövrar med en annan funktion så är den enda positionen vi behöver ändra just vid funktionsanropet, koden

```
for(int x=10; x<620; x++)
{
    double real_x = (double)(x-315)/100;
    double cosine_x = cos(real_x);
    int y = 210 - 100*cosine_x;
    DrawPixel(screen,0,255,127,x,y);
}
```

ger oss en graf över cosinus. Det enda som är ändrat är funktionsanropet. OK, vi har även infört ett annat variabelnamn (`cosine_x`), men det är bara för läslighetens skull, det förändrar inte allmängiltigheten hos tänkesättet. Vi studerar den grafen och den sammansatta koden på nästa sida.

```
//Sinus:
for(int x=10; x<620; x++)
{
    double real_x = (double)(x-315)/100;
    double sine_x = sin(real_x);
    int y = 210 - 100*sine_x;
    DrawPixel(screen,0,255,127,x,y);
}

//Cosinus:
for(int x=0; x<620; x++)
{
    double real_x = (double)(x+10-315)/100;
    double sine_x = cos(real_x);
    int y = 210 - 100*sine_x;
    DrawPixel(screen,0,255,127,x+10,y);
}
```



Övningar (troligtvis är detta några av kursens nyttigaste övningar):

Rita ut andra elementära funktioner i andra skalor, till exempel y = e ^ x (alltså e upphöjt till x.) I *C* skriver man `y = exp(x);` och den finns i biblioteket `math.h`. Rita också y = ln(x) (i *C*: y = log(x);), y = arcsin(x), y = arccos(x), y = arctan(x), y = arccot(x), y = x*x, y = 1 – x*x etc. Ta själv reda på hur man anropar arcsin, arccos, arctan, arccot etc. i *C*.

Johnny Panrike, johnny.panrike@sth.kth.se

Vi tittar nu på ett mer avancerat exempel av vad man kan använda SDL till. (Demonstrera spelet.)

En intressant sak med SDL är att det är ganska plattformsoberoende, jag skrev den här lilla grejen under Linux och flyttade den med ganska lite ansträngning till Windows.

Ni som kommer att programmera mer kommer att ha stor nytta av plattformsoberoende mjukvara, om man fördjupar sig mer i programmering så visar det sig att det är en viktig och önskvärd egenskap. Besläktat med plattformsoberoende är modularitet, vi vill skriva programvara som är isärtagbar, vi vill kunna plocka isär mjukvara och sätta samman den på nya sätt. Vi kunde tidigare se att algoritmen med att rita ut sinusgrafen var isärtagbar, vi kunde lätt lägga till consinusgrafen till samma program.

Om man går vidare i det här ska man gå till något som heter *Objektorienterad programmering*, då använder man kanske något av språken *C++* eller *Java*, men det kanske inte alla elektroingenjörer blir experter på, även om ni absolut kommer att stöta på objektoeientering. I denna kurs kommer vi absolut inte in på objektorientering och det är bra det. (Man kan ju inte klara av allt på en gång.)

Johnny Panrike, johnny.panrike@sth.kth.se