Grundläggande C-programmering – del 5
Applikationsbyggnad/Spelprogrammering (realtidsstyrsystem) och Avancerad C

Ulf Assarsson

Läromoment:

- Kodningskonventioner
- Realtidsloop
- Överkurs:
  - grafikloop, fraktalt berg
  - C99
  - obskyra C-konstruktioner
  - Dubbelpekare,

Kopplat till:

- Lab 5 - spelprogrammering

Föregående lektion

- Synlighet – static, #extern, (inline), #if/#ifdef, #include guards,

```
void testFkt()
{
    static int timesVisited = 0;
    timesVisited++;
}
```

```
void delay_500ns(void)
{
#ifndef   SIMULATOR
          delay_250ns();
          delay_250ns();
#endif
}
```

```
// player.h
#ifndef PLAYER_H
#define PLAYER_H

extern OBJECT player;

void movePlayer(Vec2i v);

#endif //PLAYER_H
```

- enum, union, little/big endian

```
enum day {monday=1, tuesday, wednesday, thursday, friday, saturday, sunday};
enum day today;
today=wednesday; // == (int)3
```

```
typedef union {
      float v[2];
      struct {float x,y;};
} Vec2f;
```

```
Vec2f pos;
pos.v[0] = 1;
pos.x = 1;
```

- Dynamisk minnesallokering (malloc/free)

```
int* p;
p = (int*)malloc( 1000 * sizeof(int));
free(p);
```

Kodningskonventioner

- Aktiveringspost (stack frame)
- Prolog
- Epilog

Varför kodningskonventioner?

- Förståelse för
  - Skillnaden mellan lokala/globala variabler.
  - Funktionsargument.
  - Returvärde.
- Möjliggör
  - Mix av assembler och C.

Detaljerna styrs av konventioner:

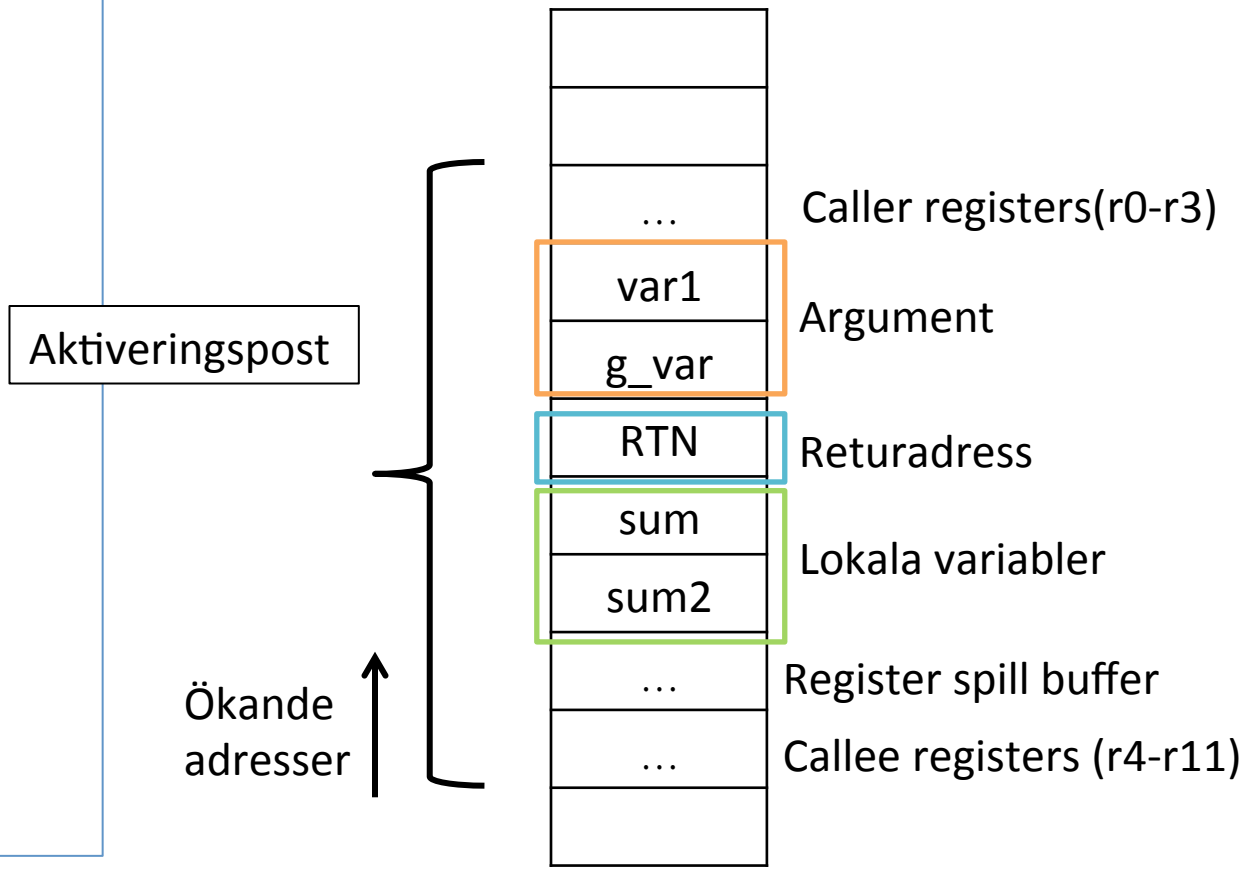
- application binary interface" (ABI) − calling conventions

Funktionsanropskonventioner
(calling conventions)

- *Parametrar, två metoder kombineras*
  - Via register: snabbt, effektivt och enkelt, begränsat antal
  - Via stacken: generellt
- *Returvärden*
  - Via register:för enkla datatyper som ryms i processorns register. T ex R0 och ev. R1
  - Via stacken: sammansatta datatyper (poster och fält)

Aktiveringspost / Stack frame – ett generaliserat exempel (ej ARM)

```
int myAdd(int x, int y)
{
    // lokala variabler
    int sum, sum2;
    sum = x+y;
    return sum;
}


var2 = myAdd(g_var, var1);
```

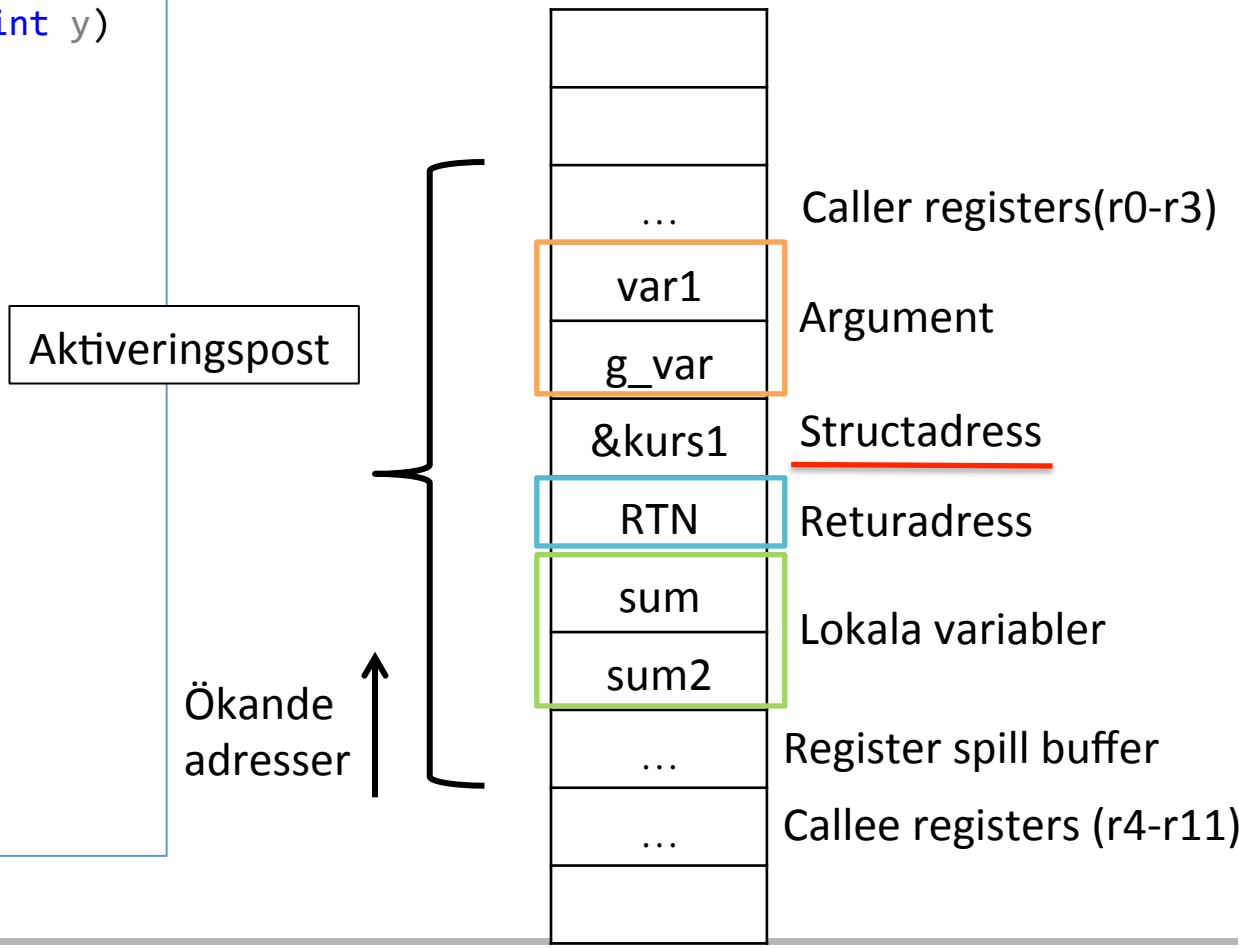


Men: ARM-Cortex M4: använder typiskt LR (r14), the link register, istället för att lägga returadressen på stacken

Aktiveringspost / Stack frame – med struct som returparameter

Returstructadressen pushas som en hemlig 0:e inputparameter.

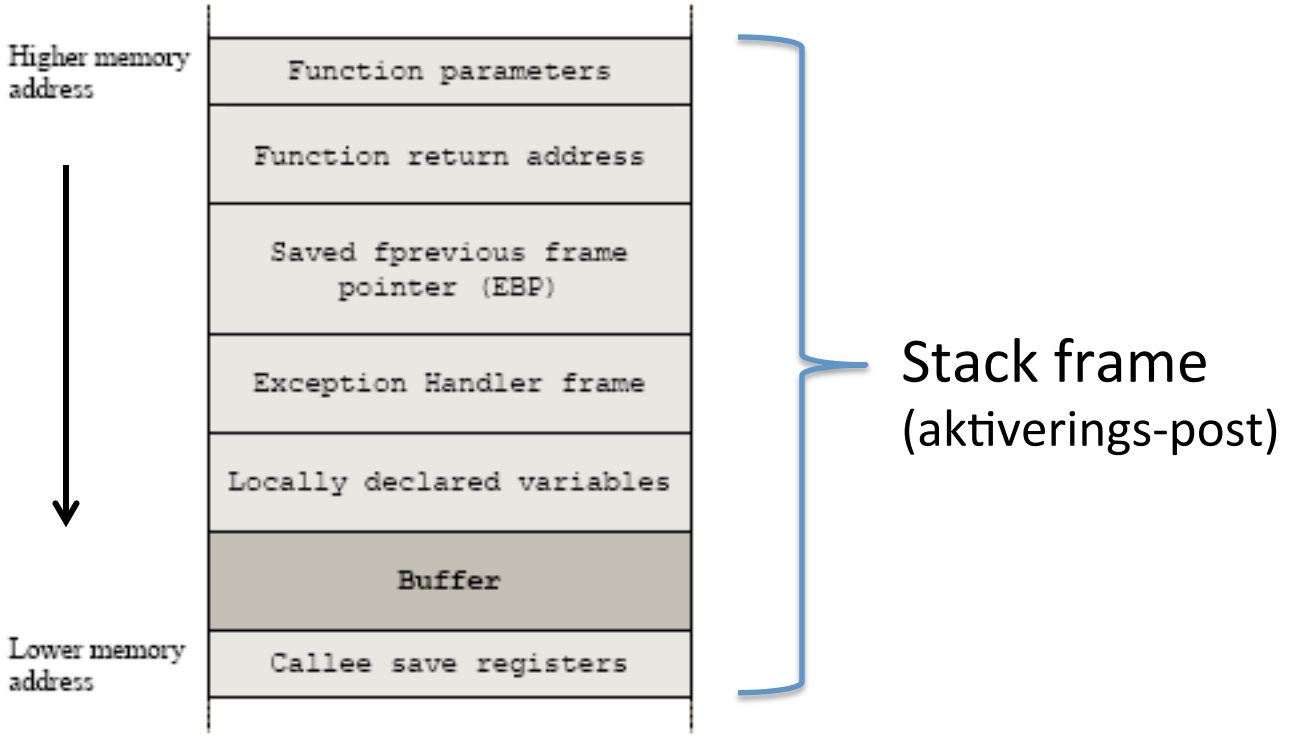
```
struct Course myFkt(int x, int y)
{
    // lokala variabler
    int sum, sum2;
    struct Course kurs;
    …
    kurs.name = "MOP";
    return kurs;
}

struct Course kurs1;
kurs1 = myFkt(g_var, var1);
```



Prolog och Epilog

- Prologen av en funktion skapar utrymmet för lokala variabler etc.
  - Görs genom att flytta (dekrementera) SP
- Epilogen av en funktion återlämnar minnet för lokala variabler etc.
  - Görs genom motsvarande tillbakaflyttning (inkrementering) av stackpekaren.

Aktiveringspost hos x86

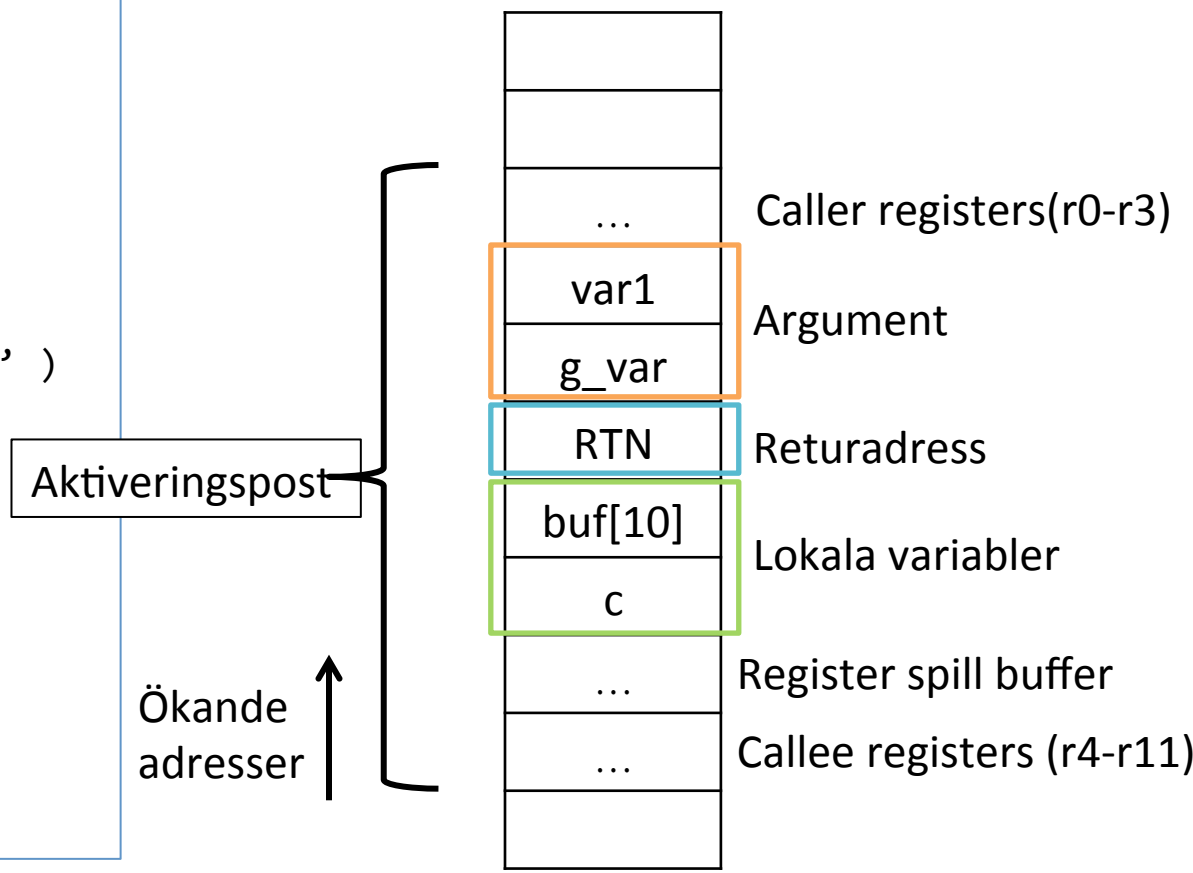


Generaliserat recept för ett funktionsanrop
(ej exakt så som ARM Cortex M4 gör)

1. Pusha argumenten på stacken.
2. ”JSR/BL” (pusha återhoppsadressen på stacken).
3. Prolog: skapar utrymme för lokala variabler etc.
4. { Funktionskroppen }
5. Lägg returvärde i rätt register eller kopiera direkt till rätt structadress.
6. Epilog: återlämnar utrymme för lokala variabler etc.
7. ”RTS/BX LR” (pop av programräknaren PC)
8. Återställ stacken till tillståndet innan argumenten pushades.

Injicera skadlig kod via stacken…

```
void input (char* s)
{
    // lokala variabler
    char buf[10], c;
    int  i=0;

    // unsafe input
    while ( (c=getch()) != '\r' )
        buf[i++] = c;
    buf[i] = '\0';
    strcpy(s, buf);
}

char s[10];
input(s);
```



När vi overflow:ar buf[10] så skriver vi över returadressen på stacken. Alltså kan vi mata in egen returadress…

```
#include <stdio.h>
#include <string.h>
#include <conio.h>

void craft_string(char *str);
void overflow();
void input ();

char s[] = "Hello World! Overwrite the return
address on your stack...\n";

int main(int argc, char **argv)
{
   printf("%s", s);
   craft_string(s);
   printf("%s", s);
   input();
   return 0;
}
```

```
void input ()
{
   // lokala variabler
   char buf[10], c;
   int  i=0;

   // unsafe input
   while ( (c=getch()) != '\r' )
      buf[i++] = c;
   buf[i] = '\0';

   strcpy(buf, s); // Overflow stack frame med vår sträng för att
}                  // därmed skriva över returadressen på stacken.

// Skapa en sträng för att injicera en annan returadress.
void craft_string(char *str) {
   // After analysis, the return PC is 24 bytes after the head of str
   // (20 bytes for 32-bits applications (Windows, Intel, gcc))
   long long *dst = (long long *)(str + 24);
   *dst = (long long) &overflow;
}

void overflow() {
   // Whatever we want to do…
   while (1)
      printf("Stack Overflow!");
}
```

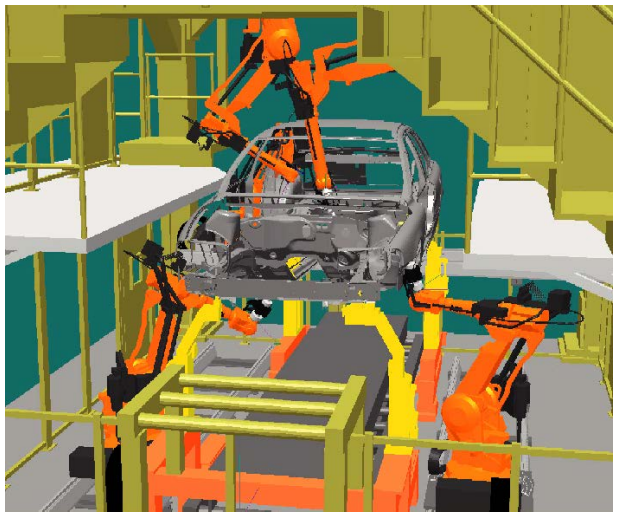
Här kan man mata in egen returadress inkl. egen kod på stacken i värsta fall.

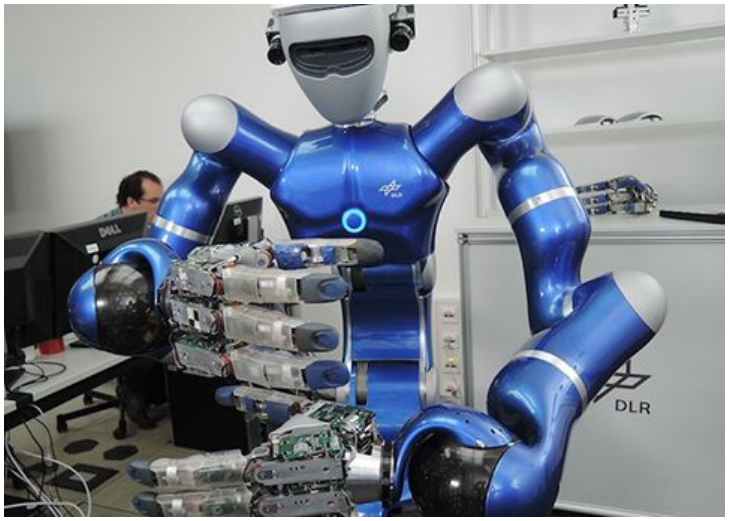
Som illustration använder jag istället strcpy för att skriva adressen till overflow() som returadress.

(OBS. Ofta tillåter dock inte ett modernt operativsystem att exekvera programkod som ligger på stacken.)

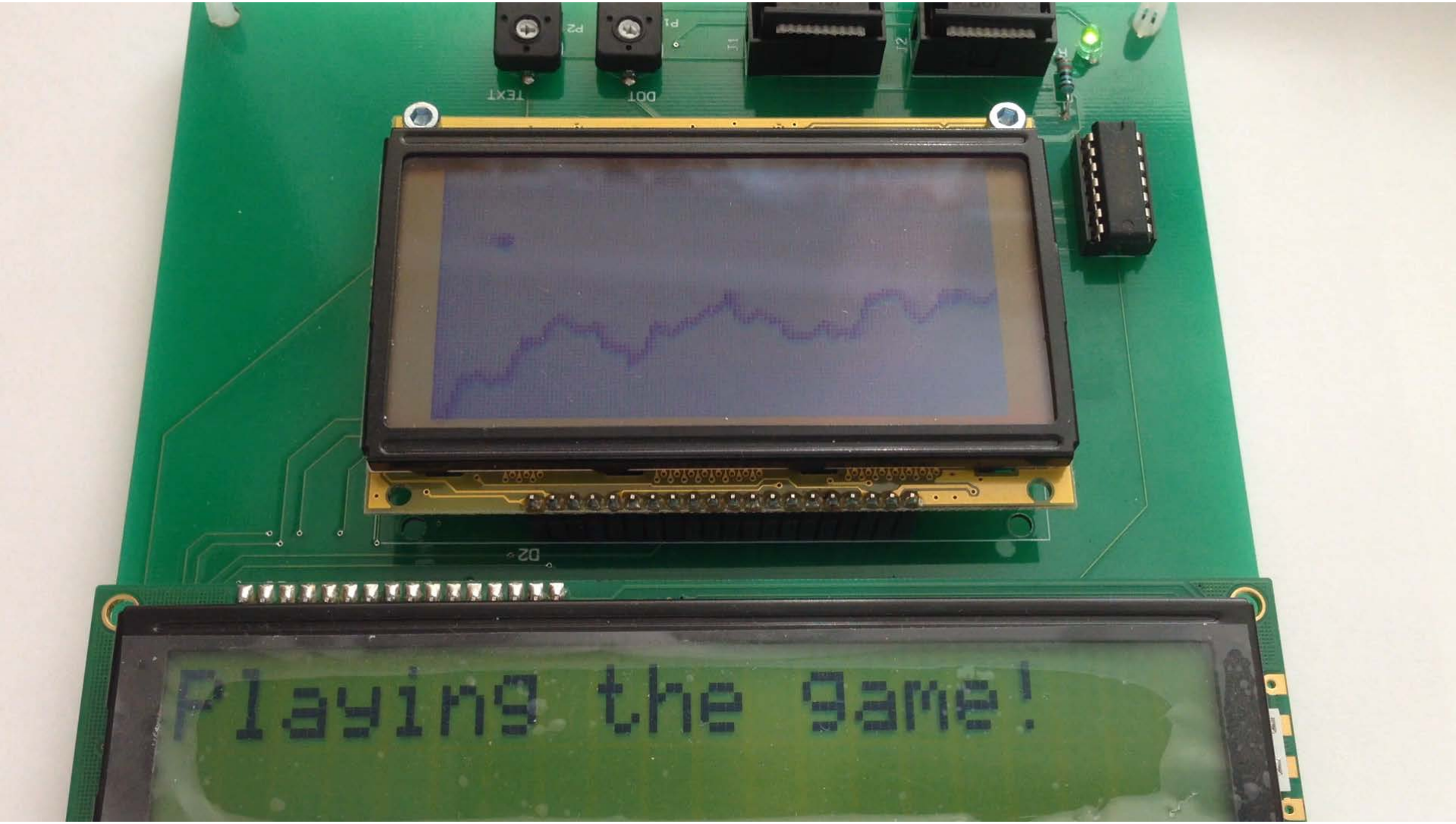
Spelprogrammering (realtidsstyrsystem)

- Realtidsspel är ofta avancerade exempel på styrsystem i realtid.
  - Jfr animera ABB-robot med att animera avatarer eller andra spelobjekt.





Ett enkelt spel…



Game loop

En typisk game loop.

```
int main(int argc, char **argv)
{
   appInit();                    // initialize GPIO_E
   graphic_initalize();          // initialize the lcd-display
   graphic_clearScreen();
   clearBuffers();               // clear front/backbuffer
   bool gameover = false;
   while( !gameover )    {
      clearBuffer(0);
      collisionDetection(); // sets flags on the objects
      updatePlayer();       // user input
      updateObjects();      // moves and/or kills objects based on the flags
      drawObjects();        // draws the objects (based on the flags)
      swapBuffers();        // draw backbuffer and swap back/frontbuffer.
      delay_milli(40);      // ~25 frames/sec
   }
}
```

Game loop

```
POBJECT objects[] = {&player, osv…};
unsigned int nObjects = 1;

void updateObjects() {
   for(int i=0; i<nObjects; i++)
      objects[i]->move(objects[i]);
}

void drawObjects() {
   for(int i=0; i<nObjects; i++)
      objects[i]->draw(objects[i]);
}

void updatePlayer() {
   switch( tstchar() ) { // tstchar() checkar input via USART-porten
      case '6': player.set_speed( &player, 2,  0); break;
      case '4': player.set_speed( &player, -2, 0); break;
      case '8': player.set_speed( &player, 0, -2); break;
      case '2': player.set_speed( &player, 0,  2); break;
   }
}
```

```
void collisionDetection()
{
    ; // insert your own code
}
```

```
static OBJECT player = {
  &ball_g,  // geometri för en boll
  0,0,      // initiala riktningskoordinater
  1,1,      // initial startposition
  draw_object,    // funktionspekare
  clear_object,
  move_object,
  set_object_speed
};
```

Game loop

En realistisk grafikloop.

```
unsigned char frameBuffer0[1024], frameBuffer1[1024];
unsigned char *frontBuffer = frameBuffer0;
unsigned char *backBuffer = frameBuffer1;

void clearBuffer(unsigned char val) {
    for (int i=0; i<1024; i++)
        backBuffer[i] = val;
}
void clearBuffers() {
    for (int i=0; i<1024; i++)
        backBuffer[i] = frontBuffer[i] = 0;
}

void swapBuffers() {
    graphic_drawScreen();
    unsigned char* tmp = frontBuffer; // swap front/backbuffers
    frontBuffer = backBuffer;
    backBuffer = tmp;
}
```

```
int main(int argc, char **argv)
{
   appInit();
   graphic_initalize();
   graphic_clearScreen();
   clearBuffers();

   bool gameover = false;

   while( !gameover ) {
      clearBuffer(0);
      collisionDetection();
      updatePlayer();
      updateObjects();
      drawObjects();
      swapBuffers();
      delay_milli(40);
   }
}
```

Motivering:
Att skriva ut på displayen är långsamt. Har man 100-tals objekt är det snabbare att istället kopiera ut en hel skärm från en framebuffer och dessutom slippa `graphic_clearScreen()`.

Game loop

En realistisk grafikloop.

```
void graphic_drawScreen(void)
{
   unsigned int k = 0;
   bool bUpdateAddr = true;
   for(uint8 c=0; c<2; c++) {                          // loop over both controllers (the two displays)
      uint8 controller = (c == 0) ? B_CS1 : B_CS2;
      for(uint8 j = 0; j < 8; j++) {                   // loop over pages
         graphic_writeCommand(LCD_SET_PAGE | j, controller );
         graphic_writeCommand(LCD_SET_ADD | 0, controller);
         for(uint8 i = 0; i <= 63; i++, k++) {         // loop over addresses
            // update display only where it is different from last frame
            if( backBuffer[k] != frontBuffer[k] ) {
               if(bUpdateAddr )
                  graphic_writeCommand(LCD_SET_ADD | i, controller);
               graphic_writeData(backBuffer[k],  controller);
               bUpdateAddr = false; // Display hardware auto-increments the address per write
            } else
                bUpdateAddr = true; // No write -> we need to update the x-address next write
            }
         }
      }
   }
}
```

Game loop

Vi får då även uppdatera pixel().

```
void pixel( int x, int y, int set )
{
   if (!set) return;

   if( (x > 127 )  || (x < 0) || (y > 63) || (y < 0) )
      return;

   unsigned char mask = 1 << (y%8);

   int index = 0;

   if(x>=64) {
      x -= 64;
      index = 512;
   }
   index += x + (y/8)*64;

   backBuffer[index] |= mask;
}
```

```
switch( y%8 )
{
        case 0: mask = 1; break;
        case 1: mask = 2; break;
        case 2: mask = 4; break;
        case 3: mask = 8; break;
        case 4: mask = 0x10; break;
        case 5: mask = 0x20; break;
        case 6: mask = 0x40; break;
        case 7: mask = 0x80; break;
}
```

Fractal mountains

```
POBJECT objects[] = {&player, &fmountain};
unsigned int nObjects = 2;
```

```
static OBJECT fmountain = {
  0,          // geometri behöver vi ej
  0,0,        // initiala riktningskoordinater
  0,0,        // initial startposition
  drawMountain,    // funktionspekare
  0,                // unused
  move_object,          // dummy
  set_object_speed    // dummy
};
```

```
static OBJECT player = {
  &ball_g,  // geometri för en boll
  0,0,        // initiala riktningskoordinater
  1,1,        // initial startposition
  draw_object,    // funktionspekare
  clear_object,
  move_object,
  set_object_speed
};
```

Fractal mountains

```
static unsigned char y_values[128];

void drawMountain() {
   static bool bFirstTime = true;
   if(bFirstTime) {
      bFirstTime = false;
      for(uint8 x=0; x<128; x++)
         y_values[x] = fractal();
   } else {
      // shift mountain to the left
      for(uint8 x=0; x<127; x++)
         y_values[x] = y_values[x+1];
      y_values[127] = fractal();
   }
   // anropa pixel för alla x=[0,127]
   for(uint8 x=0; x<127; x++)
      pixel(x, y_values[x], 1);
}
```

Fractal mountains

```
#define RAND_MAX 32767
static unsigned int next = 1;
static unsigned int rand(void)
{
   next = next * 1103515245 + 12345;
   return (unsigned int)(next/65536) % 32768;
}
static void srand(unsigned int seed) { next = seed; }
```

Rand() finns i stdlib.h, men vi saknar ju alla standardbibliotek för MD407:an, så vi måste implementera själva, t ex enligt ovan.

Fractal mountains

```
static unsigned char fractal()
{
   // f = noise(t)*A + noise(0.5t)*2A + noise(0.25t)*4A...
   static int noise[] = {0,0,0};  // room for 3 octaves
   static int t = 0;
   static int f = RAND_MAX / 2;
   const int half_max = RAND_MAX/2;
   noise[0] = (rand() > half_max) ? 1 : -1; // update each time
   if( (t%2) == 1)
      noise[1] = (rand() > half_max) ? 1 : -1; // update every 2nd time
   if( (t%4) == 3)
      noise[2] = (rand() > half_max) ? 1 : -1; // update every 4th time
   // sum octaves
   int val = 0;
   for(int i=0; i<3; i++)
      val += noise[i];
   f += val; // update our static non-bounced fractal function
   t = (t+1) % 4;
   val = f % 64;    // return a bounced value between 32 + [0-32]
   val = (val > 31) ? 63-val : val;
   return val + 32;
}
```

Non-Standard C Extensions

C – non-standard extensions

- Följande finns som extensions till C
  - Stöds typiskt inte alls i C++
  - Dessa stöds i C99 men typiskt ej i C89 / C11 (möjligen som optional för kompilaror-tillverkaren).

C – nested/local functions are compiler-optional extensions to C

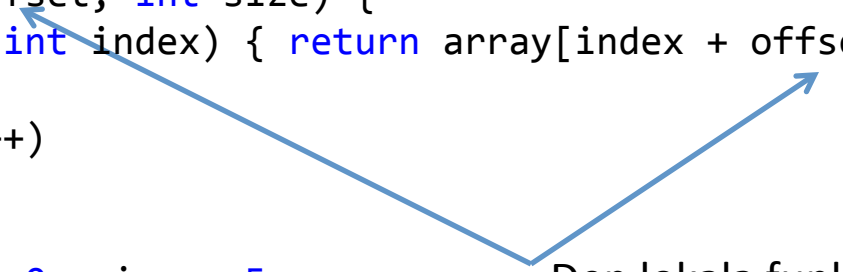
VC++ (dvs Visual Studio C++) och GNU C++ stödjer inte lokala funktioner. Det gör däremot ARM-gcc och MinGW-gcc.

```
void fkn(double a, double b)
{
  double square (double z) { return z * z; }

  return square (a) + square (b);
}
```

C – nested/local functions are compiler-optional extensions to C

VC++ (dvs Visual Studio C++) och GNU C++ stödjer inte lokala funktioner
Det gör däremot ARM-gcc och MinGW-gcc.

```
void main()
{
    …
    void fkn(int *array, int offset, int size) {
       int access (int *array, int index) { return array[index + offset]; }
       int i;
       for (i = 0; i < size; i++)
          access (array, i);
    }
    int a[] = {1,2,3,4,5}, offs=0, size = 5;
    fkn(a, offs, size);
}
```

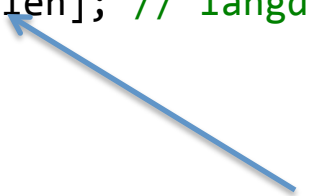


Den lokala funktionen har t.o.m. tillgång till omgivande scope's hittills deklarerade variabler.

C99 – variable-length arrays

(Optional in C11)

```
void fkn(int len)
{
    …
    char str1[10];  // längd känd i compile time.
    char str2[len]; // längd ej känd i compile time men i run time.
    …
}
```



str2 är en variable-length array (C99)

C99 – struct initiation med {.medlem, }

Initiering med { .medlem = ... } tillåter oss att endast initiera valfria medlemmar.
Exempel:

```
struct Tst {
    int a;
    char b;
};

struct Tst x1 = { .b = 'z' }; // ofullständig initiering
struct Tst x2 = { .b = 'z', .a = 5 }; // initiering i valfri ordning
struct Tst x3 = { .a = 5, .b = 'z' }; // initiering i valfri ordning
```

Luriga uttryck

Möjligt att göra – men gör inte såhär!

☺

C – luriga uttryck (överkurs)

Godtyckliga expressions.

```
for (expr/dekl; expr; expr)
{
    statement;
}

if( expr )
{
   …
}
```

Examples of statements:
`if, while, do/while, for, switch, expr`

Deklaration av lokala variabler:
`int a, b;`

Examples of expressions:

```
x = y + 3;
x++;
x = y = 0;          // Both x and y are initialized to 0
proc( arg1, arg2 );// Function call
y = z = f(x) + 3;  // A function-call expression

expr, expr;  // list of expressions. Conditional value
             // of the expression is the result of the
             // last expression.
```

C – luriga uttryck (överkurs)

```
for(expr/dekl; expr; expr)
{
    statement;
}
```

Godtyckligt expression/dekl. Dock typiskt en vanlig deklaration och initiering av loopvariabeln.

```
for (int a=1, b=0; b<5, a++, f(a), a<3; b++, a += (b>a) ? 1 : -1)
{
    a++, b++;
}
```

Godtyckligt expression.

OBS – vilkorets värde endast lika med sista uttrycket a<3 (så b<5 har här ingen effekt).

```
if(a=0, b++, a = (b == c) )
{
    …
}
```

C – luriga uttryck (överkurs)

Ett till knäppt exempel

Vad skrivs ut om man ändrar ++a till a++ ?

```
void f(int b, int c)
{
   printf("%d, %d", b, c);
}

void main()
{
   int a = 0;
   f( (++a, ++a, ++a), (++a, ++a));
}
```

Kommaseparerade expressions evalueras vä -> hö.
Så även oftast för inputparametrar (men ospecificerat).
Värdet för en lista av expr är värdet av sista uttrycket.

Expr för 1:a parametern. Expr för 2:a parametern.

Vad skrivs ut?
3, 5

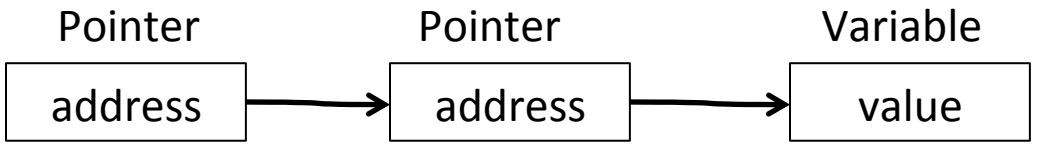
Värde = värdet hos sista uttrycken (++a).

Dubbelpekare (om vi hinner)

(pekare till pekare)

Pekare till pekare

```
char *p1, *p2, *p3;
char **pp;

pp = &p1;

// Exempel. Funktion som allokerar minne dynamiskt, t ex för elaka fiender i ett spel
void allocateEnemyArray(struct Enemy **pp, int n)
{
  *pp = (struct Enemy *)malloc(n * sizeof(struct Enemy));
}

int main()
{
  struct Enemy *pEnemies = NULL;
  allocate(&pEnemies, 100);
  …
  free(pEnemies);
}
```



pp är av typ dubbelpekare.
pp pekar på pEnemies som är av typ pekare.
*pp =… ändrar värdet för variabeln pEnemies.

Ska funktionen kunna uppdatera argumentet måste vi skicka in adressen för argumentet (istället för värdet på argumentet). Eftersom pEnemies är en pekare så är adressen till pEnemies av typen dubbelpekare (pekare till pekare till struct Enemy).

Pekare till pekare (dubbelpekare)

```
#include <stdio.h>
#include <conio.h>

char* s1 = "Emilia";  // variabeln s1 är en variabel som går att ändra, och
                      vid start tilldelas värdet av adressen till 'E':
char s2[] = "Emilia"; // värdet på s2 känt vid compile time. s2 är konstant,
                      dvs ingen variabel som går att ändra. Är adressen till 'E'.
int main()
{
        char** pp;
        char* p = s1; // == &(s1[0])
        pp = &p;
        *pp = s2; // ändrar p till s2
        **pp = 'A'; // ändrar s2[0] till 'A'
        printf("%s\n", p);

        getch();
        return 0;
}
```

Dubbelpekare. Exempel

```
int main()
{
    char s1[] = "Emilia";
    char **pp, *p;


   -- MODIFIERA s1 via dubbelpekaren pp---


    printf("s1 = %s", s1);
    getch();
    return 0;
}
```

Om ni är klara – gör trippelpekare etc…

Dubbelpekare. Lösning

```
int main()
{
    char s1[] = "Emilia";
    char **pp, *p;
    p = &s1[0];       // p = adressen till 'E'

    // eller
    p = s1;           // dvs p pekar på arrayen s1.


    pp = &p;          // pp pekar på pekaren p
    **pp = 'A’        // dubbel avreferering av "pp som pekar på p som pekar på 'E'".
    // eller
    (*pp)[0] = 'A';   // *pp pekar på p. Så det blir: ”p[0]” = 'E'

    *(*pp + 2) = 'e'; // *(p + 2). Dvs innehållet i (p + 2) tilldelas 'e’.


    printf("s1 = %s", s1);
    getch();
    return 0;
}
```