

Sammansatta datatyper

Ur innehållet

- Användardefinierade typer, "typedef"

- En komplex, användardefinierad typ "struct" ("post")

- Portbeskrivning med "struct"

- Inkapsling av data och funktioner

Läsanvisningar:

- Lärobok kap 5.4

Målsättningar:

- Kunna använda sammansatta typer för inkapsling ("objekt")

- Kunna programmera enkla geometrier för grafisk display

typedef - alias för en typ

typedef används för att skapa ett *alias*, oftast för att förenkla och förkorta typuttryck. Avsikten är att förtydliga och öka läsbarheten, syntaxen är:

typedef *typ* *alias_typnamn* [, *alias_typnamn*]... ;

Exempel:

```
typedef unsigned char    uint8;
typedef short int        int16;
// Anm. '*' är en del av alias-namnet och ingår INTE i befintlig typ
typedef unsigned char    *ucharptr;

                typ          alias_typnamn

uint8 a, b = 0, c;
ucharptr p;
typedef int postnr;
typedef int strtNr;
postnr x = 41501;
strtNr y = 3;
x = y; // Typriktigt, helt ok
```

struct ("post"), en sammansatt datatyp

- Har en eller flera medlemmar (*fields*).
- En medlem kan vara av godtycklig typ, exempelvis:
`int`, `char`, `long` (`signed/unsigned`), `float`, `double`
- Användardefinierad (med "`typedef`")
- Sammansatt typ (dvs. en annan `struct`).
- Alla typer av pekare

Deklarationssyntax:

```
struct structnamn{  
    medlem;  
    [medlem; [medlem;]... ]  
};
```

Deklaration av "struct" (post)

```
// Deklaration av structens typ
struct structnamn{
    ...
};

// Deklaration av structens typ och
// en variabel av denna typ
struct structnamn{
    ...
} variable;
```

```
// Det är vanligt med användardefinierade struct-typer
typedef struct structnamn TSTRUCTNAMN;
// alternativt:
typedef struct structnamn /* structnamn kan utelämnas här */
{
    ...
} TSTRUCTNAMN;
```

Exempel:

```
// Datatyp för koordinater
typedef struct coord
{
    int x,y;
} COORD;
// Variabeldeklarationer
struct coord start,end;
//eller..
COORD start, end;
```

Användning

```
#include <stdio.h>

typedef struct coord{ int x,y; } COORD;

int main()
{
    struct coord start;
    COORD end;

    start.x = 10; start.y = 10;
    end.x   = 20; end.y   = 20;

    printf("Line starts at (x,y) %d,%d \n
           and ends at (x,y) %d,%d \n",
           start.x, start.x,
           end.x, end.x);

    return 0;
}
```

Typen kan användas på två sätt:
`struct coord` eller
`COORD`

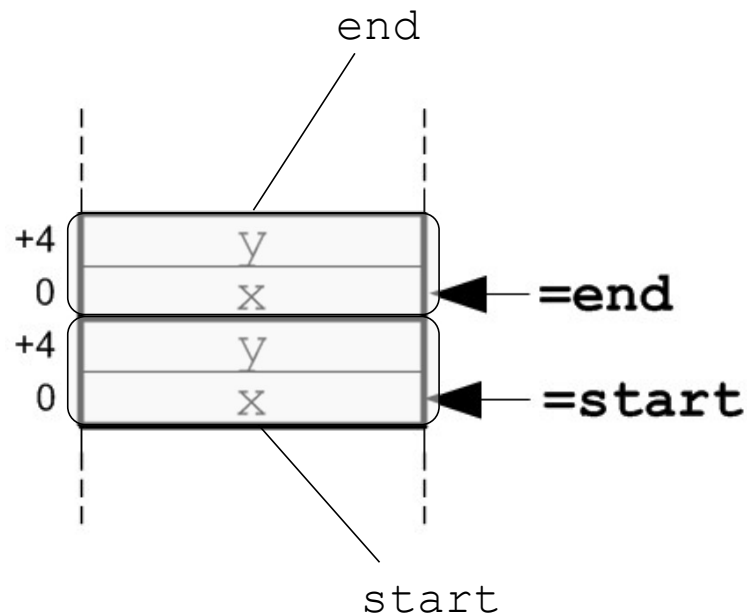
Likvärdiga deklarationer av
variablerna `start` och `end`

Medlemmarna refereras via `"."`-
operatorn ("punktoperator")

Referenser

Exempel: Datatyp för koordinater

```
typedef struct
{
    int x, y;
} COORD;
```



Exempel: Vi har deklarationerna:

```
COORD start, end;
```

Koda tilldelningen:

```
start.y = end.y;
```

i ARM/Thumb assemblerspråk

Lösning:

```
LDR R0, =end      @ R0 ← &end
LDR R0, [R0, #4]  @ R0 ← end.y
LDR R1, =start    @ R1 ← &start
STR R0, [R1, #4]  @ start.y = end.y
```

Inbäddad post

- Ev. rättningsvillkor måste respekteras.
- En annan post kan ingå i en större post.

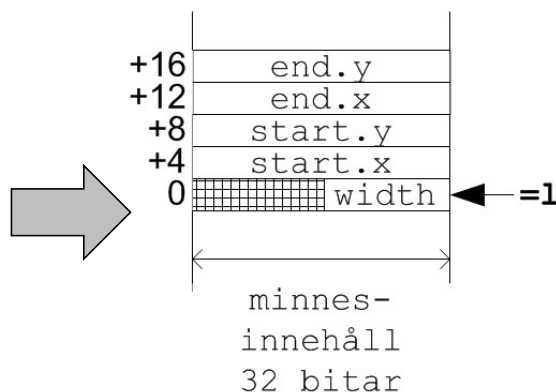
Exempel: Datatyp för koordinater

```
typedef struct
{
    int x, y;
} COORD;
```

Vi har deklarationen:

```
typedef struct {
    short width;
    COORD start;
    COORD end;
} LINE, *PLINE;
LINE l;
```

Exempel:



```
.ALIGN    2
offset_width = 0
offset_start.x = offset_width + sizeof(width) + align(2)
offset_start.y = offset_start.x + sizeof(start.x) + align(2)
offset_end.x = offset_start.y + sizeof(start.y) + align(2)
offset_end.y = offset_end.x + sizeof(end.x) + align(2)
```

Visa kodsekvenser som
evaluerar följande uttryck i
register R0.

- l.width
- l.end.y

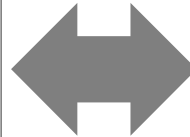


```
a)
LDR    R0, =1
LDRH   R0, [R0, #0]      @ R0= 1+ 0 (offset_width)
b)
LDR    R0, =1
LDR    R0, [R0, #16]     @ R0= 1+ 16 (offset_end.y)
```

Alternativa deklarationer

```
struct line
{
    struct coordinate
    {
        int x,y;
    } start, end;
} linje;
// Referenser
linje.start.x;
linje.end.y; // etc
```

`struct coordinate`
kan bara användas som medlem av
`struct line`.



```
struct coordinate
{
    int x,y;
};

struct line
{
    struct coordinate start;
    struct coordinate end;
} linje;
// Referenser
linje.start.x;
linje.end.y; // etc
```

`struct coordinate`
och
`struct line`
kan användas var och en för sig.

Initiering, fullständig och ofullständig

En `struct` kan initieras med en lista.

Medlemmarna tilldelas värden i samma ordning som deklarationen.

Listan kan vara ofullständig, dvs. alla medlemmar behöver inte initieras.

```
struct Course {  
    char* name;  
    float credits;  
    int numberOfParticipants;  
};  
  
struct Course c1 = {"MOP", 7.5, 110};  
struct Course c2 = {"MOP", 7.5};
```

Fullständig initiering

Ofullständig initiering

Ofullständig deklaration

En struct kan först ges som en ofullständig deklaration, för att senare definieras fullständigt.

Exempel:

```
struct coordinate; /* coordinate är en struct */
```

```
struct line
{
    struct coordinate start;
    struct coordinate end;
};
```

```
struct linepointers
{
    struct coordinate *start;
    struct coordinate *end;
};
```

Fel:

Storleken av `struct coordinate` är okänd, alltså kan inte storleken av `struct line` bestämmas.

Ok:

En pekarstorlek är alltid känd, alltså kan storleken av `struct linepointers` bestämmas.

Pekare till **struct**, pilnotation

Det är vanligt att använda pekare och pilnotationen förenklar detta betydligt.

Exempel:

```
struct coordinate {int x,y;} c1;
struct coordinate *ptr;

ptr = &c1;
// Direkt, variabel via punktoperator
c1.x = ...
// eller via derefererad pekare
(*ptr).x = ...
// eller kortare, piloperatorn:
p->x    = ...
```

Exempel: Koordinater i en länkad lista

```
typedef struct tPoint{
    char x;
    char y;
    struct tPoint *next;
} POINTLIST;

/* Gå igenom lista av koordinater... */
POINTLIST *first; /* pekar på listan */
...
POINTLIST *p = first;
while( p != (POINTLIST *) 0 )
{
    /* Gör något med koordinaten ... */
    p = p->next;
}
```

Pekare till struct, pilnotation

Exempel:

```
typedef struct coord
{
    int x,y;
    struct coord *next;
} COORD;
COORD c1, ... , c10;
COORD *first, *p;
```

Koda tilldelningarna:

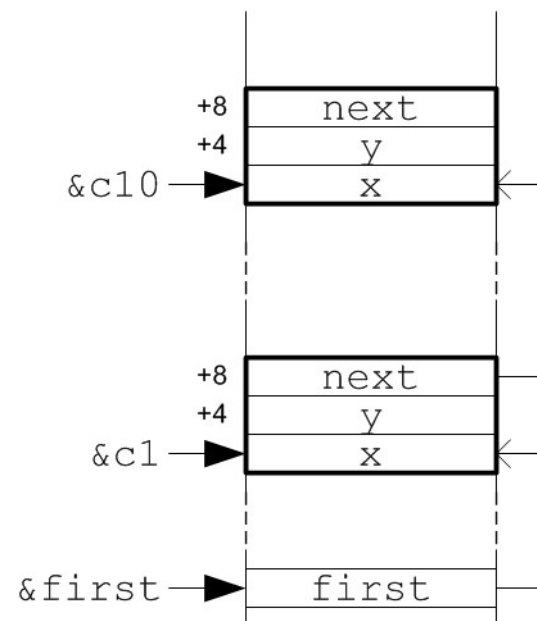
```
first = &c1;
p = first;
p->next = &c10;
```

*(*p).next = &c10;*
p->next = &c10;

```
@ first = &c1;
LDR    R0,=c1
LDR    R1,=first
STR    R0,[R1]
```

```
@ p = first;
LDR    R0,=first
LDR    R0,[R0]
LDR    R1,=p
STR    R0,[R1]
```

```
@ p->next;
LDR    R0,=c10
LDR    R1,=p
LDR    R1,[R1]
STR    R0,[R1,#8]
```



Portadressering med poster

struct-typen är också användbar för att deklarerar portar

offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Register
0																																	GPIO_MODER
4																																	GPIO_OTYPER
8																																	GPIO_OSPEEDR
0xC																																	GPIO_PUPDR
0x10																																	GPIO_IDR
0x14																																	GPIO_ODR
0x18																																	GPIO_BSRR
0x1C																																	GPIO_LCKR
0x20																																	GPIO_AFR1
0x24																																	GPIO_AFRH

// Som alternativ till :

```
#define portModer ((volatile unsigned int *) (GPIO_BASE))
#define portOtyper ((volatile unsigned int *) (GPIO_BASE+0x4))
#define portOspeedr ((volatile unsigned int *) (GPIO_BASE+0x8))
osv.
```

// kan vi använda en post-definition som:

```
typedef volatile struct {
    unsigned int moder;
    unsigned int otyper; // +0x4
    unsigned int ospeedr; // +0x8
    unsigned int pupdr; // +0xC (12)
    unsigned int idr; // +0x10
    unsigned int odr; // +0x14
    unsigned int bsrr; // +0x18
    unsigned int lckr; // +0x1C
    unsigned int afr1; // +0x20
    unsigned int afrh; // +0x24
} GPIO, *PGPIO;
```



Exempel:

```
#define GPIO_D (*(volatile PGPIO) 0x40020c00)
#define GPIO_E (*(volatile PGPIO) 0x40021000)
```

```
GPIO_E.moder = 0x55555555;
GPIO_E.otyper = 0x00000000;
GPIO_D.moder = 0x55550000;
GPIO_D.pupdr |= 0x00005555;
```

Portadressering med poster

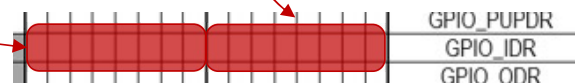
Deklarationen kan i stället anpassas till portens olika registers storlekar

```
typedef volatile struct {
    unsigned int  moder;
    unsigned int  otyper;
    unsigned int  ospeedr;
    unsigned int  pupdr;
    unsigned int  idr;
    unsigned int  odr;
    unsigned int  bsrr;
    unsigned int  lckr;
    unsigned int  afrl;
    unsigned int  afrh;
} GPIO, *PGPIO;
```

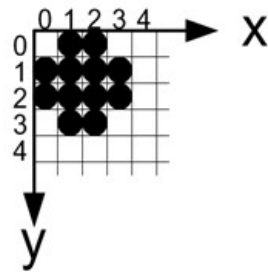
```
typedef volatile struct {
    unsigned int      moder;
    unsigned short int otyper;
    unsigned short int Reserved0;
    unsigned int      ospeedr;
    unsigned int      pupdr;
    unsigned char     idrLow;
    unsigned char     idrHigh;
    unsigned short int Reserved1;
    unsigned char     odrLow;
    unsigned char     odrHigh;
    unsigned short int Reserved2;
    unsigned int      bsrr;
    unsigned short int lckr;
    unsigned short int Reserved3;
    unsigned int      afrl;
    unsigned int      afrh;
} GPIO, *PGPIO;
```

Konvertering till lämplig storlek:

```
GPIO GPIO_E;
typedef unsigned char uint8_t;
uint8_t x = *((uint8_t*)&GPIO_E.idr;
uint8_t y = *((uint8_t*)&GPIO_E.idr+1);
```



Inkapsling av data, ett "objekt"



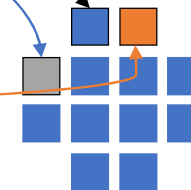
```
typedef struct tPoint{
    unsigned char x;
    unsigned char y;
} POINT;

#define MAX_POINTS 20

typedef struct tGeometry{
    int    numpoints;
    int    sizex;
    int    sizey;
    POINT  px[ MAX_POINTS ];
} GEOMETRY, *PGEOMETRY;
```

Exempel: Arbetsboken, avsnitt 5.4

```
// Skapa och initiera ett object av typen GEOMETRY:
GEOMETRY ball_geometry = {
    12, 4, 4, // numpoints, sizex, sizey
    { // POINT px[20]
        {0,1},
        {0,2},
        {1,0},
        {1,1},
        {1,2},
        {1,3},
        {2,0},
        {2,1},
        {2,2},
        {2,3},
        {3,1},
        {3,2} // Ofullständig initiering
    } // (12 av 20)
};
```



Poster med funktionspekare

Inkapsling av pekare till funktioner.

Exempel:

```
typedef struct tObj {  
    PGEOMETRY geo;  
    int dirx, diry;  
    int posx, posy;  
    void (*draw)(struct tObj *);  
    void (*clear)(struct tObj *);  
    void (*move)(struct tObj *);  
    void (*set_speed)(struct tObj *, int, int);  
} OBJECT, *POBJECT;
```

`draw, clear, move` och `set_speed`
är pekarter.

Kan uppfattas som
"metoder" i
objektorienterat språk.

Poster med funktionspekare

Man kan ge objektet dess initiala egenskaper med statisk initiering

Exempel statisk initiering:

```
OBJECT ball = {  
    &ball_geometry,  
    0, 0,  
    64, 32,  
    draw_object,  
    clear_object,  
    move_object,  
    set_object_speed  
};
```

```
typedef struct tObj {  
    PGEOMETRY geo;  
    int dirx, diry;  
    int posx, posy;  
    void (*draw)(struct tObj *);  
    void (*clear)(struct tObj *);  
    void (*move)(struct tObj *);  
    void (*set_speed)(struct tObj *, int, int);  
} OBJECT, *POBJECT;  
  
void move_object(POBJECT o ) {...}  
void draw_object(POBJECT o ) {...}  
void clear_object(POBJECT o ) {...}  
void set_object_speed(POBJECT o, int x, int y ) {...}
```

...medan dynamisk initiering sker under programmets gang.

Exempel dynamisk initiering:

```
OBJECT ball;  
ball.geo = &ball_geometry;  
ball.dirx = 0; ball.diry = 0;  
ball.posx = 64; ball.posy = 32;  
ball.draw = draw_object;  
ball.clear = clear_object;  
ball.move = move_object;  
ball.set_speed = set_object_speed;
```

Poster med funktionspekare

Dynamisk initiering kan exempelvis användas för att ändra egenskaper hos ett objekt baserat på olika händelser.

```
typedef struct tObj {
    PGOMETRY geo;
    int dirx, diry;
    int posx, posy;
    void (*draw)(struct tObj *);
    void (*clear)(struct tObj *);
    void (*move)(struct tObj *);
    void (*set_speed)(struct tObj *, int, int);
} OBJECT, *POBJECT;

void move_object(POBJECT o) {...}
void draw_object(POBJECT o) {...}
void clear_object(POBJECT o) {...}
void set_object_speed(POBJECT o, int x, int y) {...}
```

Exempel: alternativ "move"-funktion:

```
// Vi definierar en alternativ move-funktion
void alternate_move_object(POBJECT o) { ... }

POBJECT pb;
pb = &ball;
pb->move = move_object;
...
pb->move( pb ); // Här utförs move_object
...
pb->move = alternate_move_object;
...
pb->move( pb ); // Här utförs alternate_move_object
...
```

Objektorientering i ett icke-objektorienterat språk

- Ideer kring inkapsling/objektorientering har utvecklats sedan 1950-talet
- C++** är ett objektorienterat språk, inkapsling finns exempelvis som konstruktioner i språket.
- Lokala funktioner tillsammans med data, konceptet kallas "klass" - en klass kan innehålla både variabler och funktioner.
- C har inte klasser, men vi kan simulera en klass med hjälp av sammansatta typer.

Inkapsling: klass i C++

```
typedef struct {  
    int a;          // a member  
    void inc() {    // method that increments a  
        a++;  
    }  
} MyClass;  
  
MyClass v = {0}; // initialize variable  
v.inc();         // increment variable
```

Inte i C...

...men vi kan åstadkomma samma sak:

```
typedef struct tMyClass{  
    int a;  
    void (*inc) (struct tMyClass* this);  
} MyClass;  
  
void incr(MyClass* this)  
{  
    this->a++;  
}  
  
MyClass v = {0, incr};  
v.inc(&v);
```

Observera!

Objektorientering med C

Detta...

```
typedef struct tMyClass {
    int    a;
    void (*inc) (struct tMyClass* this);
} MyClass;
```

`tMyClass` används för att ange parameterns typ eftersom `MyClass` fortfarande är odefinierad.

... är ekvivalent med detta:

```
struct MyClass;
typedef struct {
    int    a;
    void (*inc) (MyClass* this);
} MyClass;
```

Ofullständig deklaration av `MyClass`, den fullständiga deklarationen ges av den sista raden...

..men en pekare behövs här

Båda metoderna fungerar...

Hur metदानropet `.inc(...)` fungerar i C:

```
// MyClass kallar vi objekt.
// v1, v2 är två instanser av objektet.
MyClass v1 = {0, incr}, v2 = {1, incr};
// Metदानrop inc() för v1 och v2.
v1.inc(&v1);
v2.inc(&v2);
```

```
// incr() är en vanlig function
// som både v1.inc och v2.inc pekar på.
void incr(MyClass* this)
{
    this->a++;
}
```

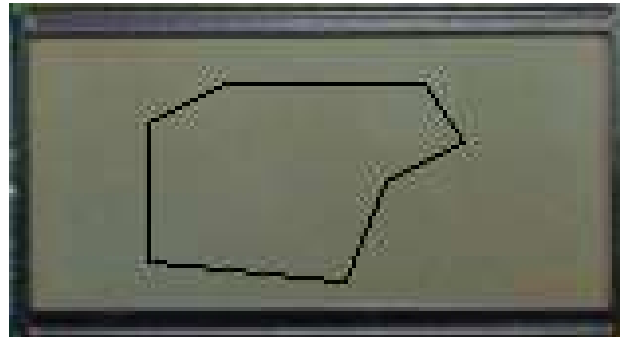
Initieringarna gör att `v1.a = 0`, `v2.a = 1`, `v1.inc` och `v2.inc` pekar på funktionen `incr()`. Minns att `incr` bara är en symbol för den minnesadress där funktionen `incr()` är placerad och `v1.inc` och `v2.inc` är pekarvariabler som nu innehåller denna adress.

Här har vi anropen av de funktioner som pekarvariablerna anger med `&v1` respektive `&v2` som parametrar. Så `v1.inc(&v1)` är nu samma sak som anropet `incr(&v1)` och `v2.inc(&v2)` är samma sak som anropet `incr(&v2)`.

Anropet `incr(&v1)` betyder att parametern `this` utgör adressen till `v1`. Alltså har vi att `this->a++` är samma som `(&v1)->a++` (vilket är ekvivalent med `v1.a++`)..... vilket är precis det vi vill göra med `v1.inc(&v1)`; Det samma gäller `v2.inc(&v2)`; Dvs. inkrementering av `v2.a`.

Programmering av grafisk display

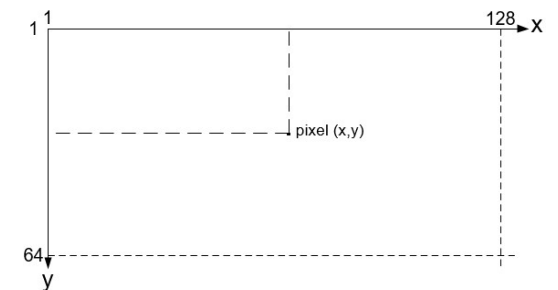
PONG



Plot av polygon



Spindeljakt



Inbyggda funktioner för grafisk display

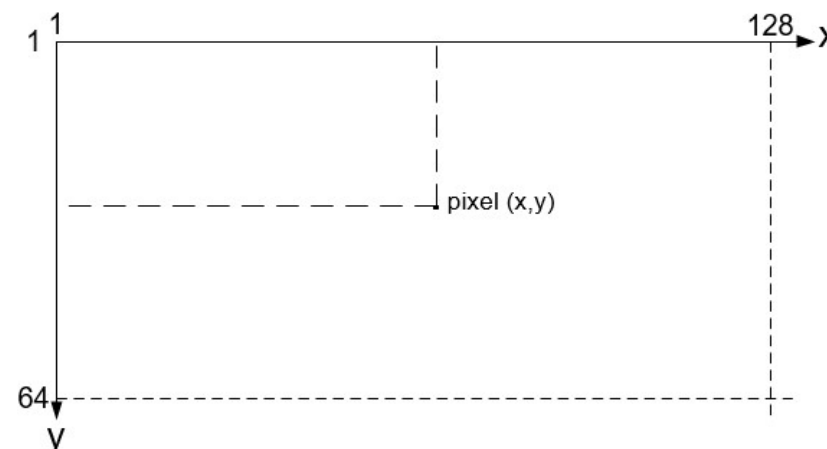
Funktioner som anropas via processorns undantagshantering

```
void graphic_initialize (void);
void graphic_clear_screen (void);
void graphic_pixel_set (int x, int y);
void graphic_pixel_clear (int x, int y);
```

Implementering:

<pre>__attribute__((naked)) void graphic_initialize(void) { __asm volatile (" .HWORD 0xDFF0\n"); __asm volatile (" BX LR\n"); }</pre>	<pre>__attribute__((naked)) void graphic_clear_screen(void) { __asm volatile (" .HWORD 0xDFF1\n"); __asm volatile (" BX LR\n"); }</pre>
<pre>__attribute__((naked)) void graphic_pixel_set(int x, int y) { __asm volatile (" .HWORD 0xDFF2\n"); __asm volatile (" BX LR\n"); }</pre>	<pre>__attribute__((naked)) void graphic_pixel_clear(int x, int y) { __asm volatile (" .HWORD 0xDFF3\n"); __asm volatile (" BX LR\n"); }</pre>

Implementeringsdetaljerna beskrivs senare i kursen...

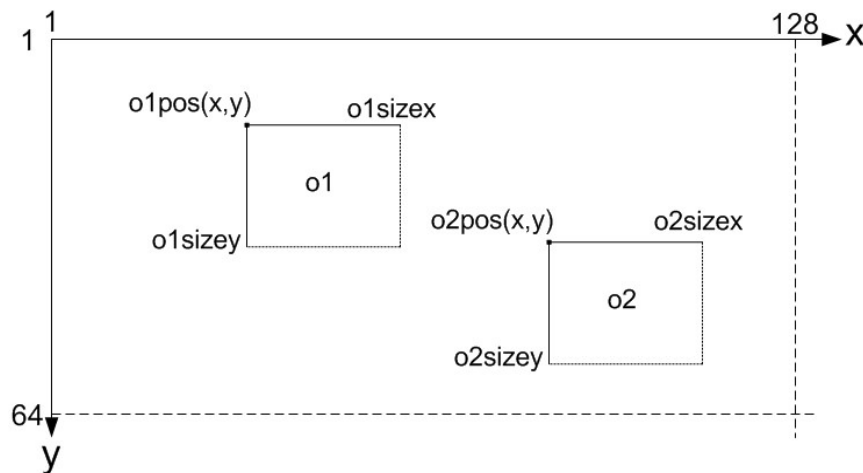


Överlappande objekt

Efter varje förflyttning av något objekt kontrolleras om de är i kontakt (överlappar).

Det intuitivt första försöket kan vara att jämföra de pixlar de respektive objekten ockuperar, stor komplexitet.

En approximativ metod kan användas för att jämföra objektens maximala utsträckning, liten komplexitet.



Metoderna kan kombineras:

```
if ( approx_overlap )  
    if( exact_overlap )  
        --objekten överlappar
```