
Data structures

So far we have mostly used built-in types

- integers, 1
- floating point numbers, 3.14

⇒ Pretty boring.

We have seen one example of something bigger:

- Arrays.

Programming languages allow you to define more types:

- Enumerated data
- Making your own types
- Flatly structured data types
- Dynamic data types.

Enumerated types, implementation

Enumerated types represent some collection:

```
typedef enum { Jan, Feb, Mar, Apr, May, Jun, Jul  
             Aug, Sep, Oct, Nov, Dec } Month;
```

Defines a type `Month` with 12 values, `Jan`, `Feb`, ...

Compare with a type `int` with 4 billion values, 0, 1, ...

Or with a type `char` with 256 values, `'a'`, `'!'`, ...

Enumerated types are represented by integers:

- first one is represented by 0, the second one by 1, ...
- You can specify something explicitly, for example:

```
typedef enum { Sunday=0, Saturday=6 } WeekendDay ;
```

Types are the sets of maths

Discrete Maths: Sets

- Standard sets:
 - Z, N, R,
- Make your own:
 - Monkeys: {gorilla,baboon}
 - Piece: {WhitePawn, WhiteKnight, ...}

Programming: Types

- First three are predefined:
 - `int`, `unsigned int`, `double`.
- You define your own:
 - `typedef enum {gorilla,baboon} monkeys ;`
 - `typedef enum {WhitePawn,WhiteKnight, ...} Piece ;`

Other types

Arrays:

```
typedef double Image\n[1280][1024] ;
```

defines a type `Image` which is an 1280×1024 array of doubles.

Can use this as follows:

```
int displayimage( Image x ) {  
    ...  
}
```

format of typedef:

```
typedef enum { ... } TYPENAME ;  
typedef type TYPENAME[ ... ] ;  
typedef type TYPENAME[ ... ][ ... ] ;  
typedef type TYPENAME[ ... ][ ... ][ ... ] ;
```

You can use `TYPENAME` just like any other type

Simple data structure

Simple data structures *group* values, aggregate data.

Example:

A date is a day,
month and year

8

May

1896

Simple data structure

Simple data structures *group* values, aggregate data.

Example:

A date is a day,
month and year



Simple data structure

Simple data structures *group* values, aggregate data.

Example:

A date is a day,
month and year



A name consists of a
first name and a surname

"Igor Fyodorovich"

"Stravinsky"

Simple data structure

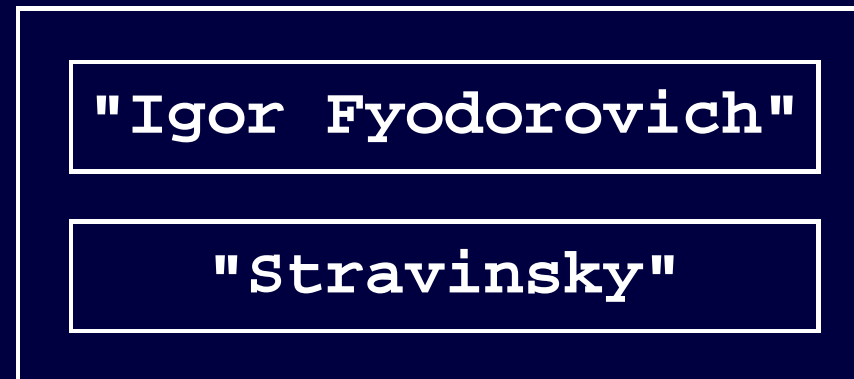
Simple data structures *group* values, aggregate data.

Example:

A date is a day,
month and year



A name consists of a
first name and a surname



Simple data structure

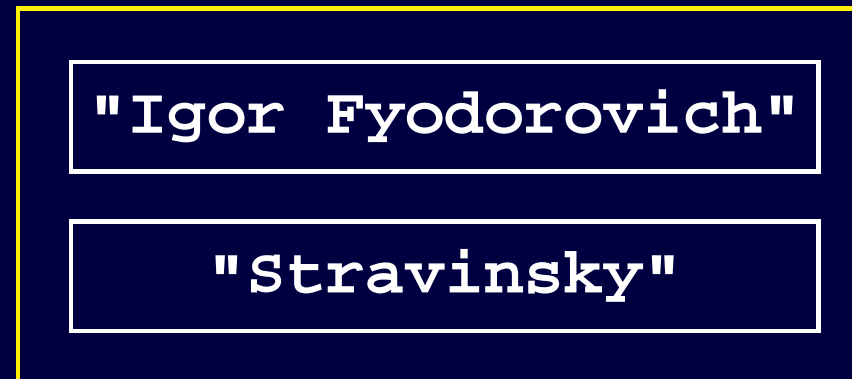
Simple data structures *group* values, aggregate data.

Example:

A composer has a name and a birthday

A date is a day,
month and year

A name consists of a
first name and a surname



Simple data structure

Simple data structures *group* values, aggregate data.

Example:

A composer has a name and a birthday

A date is a day,
month and year

A name consists of a
first name and a surname



The type of aggregate data

The type of a date is

$$N \times \text{Month} \times N$$

The type of a name is

$$\text{String} \times \text{String}$$

The type of a composer is

$$\text{name} \times \text{date}$$

or

$$(\text{String} \times \text{String}) \times (N \times \text{Month} \times N)$$

Aggregate data in C

Use the `struct` keyword to define a structure type.

```
typedef enum { Jan=1, Feb, Mar, Apr, ..., Dec } Month ;
typedef struct {
    int day ;
    int year ;
    Month month ;
} Birthday ;
```

This defines

- Types called `Month` and `Birthday`
 - The type *Birthday* is a `struct` with three members.
 - Two members are of type `int`, one is of type `Month`.

Aggregate data in C

Use the `struct` keyword to define a structure type.

```
typedef enum { Jan=1, Feb, Mar, Apr, ..., Dec } Month ;  
typedef struct {  
    int day ;  
    int year ;  
    Month month ;  
} Birthday ;
```

This defines

- Types called `Month` and `Birthday`
 - The type *Birthday* is a `struct` with three members.
 - Two members are of type `int`, one is of type `Month`.

Aggregate data in C

Use the `struct` keyword to define a structure type.

```
typedef enum { Jan=1, Feb, Mar, Apr, ..., Dec } Month ;
typedef struct {
    int day, year ;

    Month month ;
} Birthday ;
```

This defines

- Types called `Month` and `Birthday`
 - The type *Birthday* is a `struct` with three members.
 - Two members are of type `int`, one is of type `Month`.

Full example

```
typedef enum { Jan=1, Feb, Mar, Apr, ..., Dec } Month ;
typedef struct {
    int day, year ; Month month ;
} Birthday ;
```

```
int main( void ) {
    Birthday stravinsky ;
    stravinsky.day      = 8 ;
    stravinsky.month    = May ;
    stravinsky.year     = 1896 ;
    printf( "%d\n", stravinsky.month ) ;
}
```

The membership operator “.” is used to select a field

Full example

```
typedef enum { Jan=1, Feb, Mar, Apr, ..., Dec } Month ;
typedef struct {
    int day, year ; Month month ;
} Birthday ;
Month month_of_birth( Birthday b ) {
    return b.month ;
}
int main( void ) {
    Birthday stravinsky ;
    stravinsky.day      = 8 ;
    stravinsky.month    = May ;
    stravinsky.year     = 1896 ;
    printf( "%d\n", month_of_birth( stravinsky ) ) ;
}
```

You can pass a struct to a function...

Full example

```
typedef enum { Jan=1, Feb, Mar, Apr, ..., Dec } Month ;
typedef struct {
    int day, year ; Month month ;
} Birthday ;
Month month_of_birth( Birthday b ) {
    return b.month ;
}
int main( void ) {
    Birthday stravinsky = { 8, 1896, May } ;

    printf( "%d\n", month_of_birth( stravinsky ) ) ;
}
```

When you initialise a struct you can use {}

So...

Grouped data:

- Need to define a type, with a name
 - List the types that we are going to store inside
- Need some kind of constructor
 - An operation to create a tuple of that type
- Need an operation to look inside
 - Use the ‘.’ operator in C

A slightly more complicated example

Lets define a type point, in the plane XY , consisting of two reals:

$$\text{Point} = R \times R$$

If (x, y) is a point, then the point rotated with an angle ϕ is given by

$$(x \cos \phi + y \sin \phi, y \cos \phi - x \sin \phi)$$

Define a function rotate:

$$\text{rotate}((x, y), \phi) = (x \cos \phi + y \sin \phi, y \cos \phi - x \sin \phi)$$

Rotate with C structures

```
typedef struct {  
    double x, y ;  
} Point ;
```

Rotate with C structures

```
typedef struct {  
    double x, y ;  
} Point ;
```

```
Point rotate( Point s, double phi ) {  
    Point t ;  
    double sin_phi = sin( phi ) ;  
    double cos_phi = cos( phi ) ;  
    t.x = s.x * cos_phi + s.y * sin_phi ;  
    t.y = s.y * cos_phi - s.x * sin_phi ;  
    return t ;  
}
```

Union types

A Union-type (also known as a variant) allows you to store either X or Y in a data type. Use `union` in C.

```
typedef struct {  
    double d, alpha ;  
} Polar ;
```

```
typedef struct {  
    double x, y ;  
} Cartesian ;
```

```
typedef union {  
    Polar p ;  
    Cartesian c ;  
} Point ;
```

Alternatives

```
typedef union {  
    Polar p ;  
    Cartesian c ;  
} Point ;
```

Defines a data type `Point`, which consists of

- Either a cartesian part (with two floats),
- Or a polar description (with two floats).

(or, for example, a vehicle which is a car, truck, or caravan)

A member of a union is accessed with the membership operator `'.'`
(The members of a struct are accessed using `'.'`)

So:

- if `s` is of type `Point`,
- then `s.c.x` refers to member `x` of member `c` of `s`

Alternatives

```
int main( void ) {  
    Point s, t ;  
    s.c.x = 2 ;  
    s.c.y = 1 ;  
    t.p.d = 2.236 ;  
    t.p.alpha = 0.4636 ;  
}
```

s specifies the point (2,1) using Cartesian Coordinates

- 2 along the X axis, 1 along the Y axis

t specifies the point (2,1) using Polar Coordinates

- An angle of 0.4636 radians, a distance of 2.236 ($\sqrt{5}$)

Difference between Struct and Union

```
typedef union {
    Polar p ;
    Cartesian c ;
    int i ;
} uniontype ;

uniontype u ;

typedef struct {
    Polar p ;
    Cartesian c ;
    int i ;
} structtype ;

structtype s ;
```

Difference between Struct and Union

```
typedef union {  
    Polar p ;  
    Cartesian c ;  
    int i ;  
} uniontype ;
```

```
uniontype u ;
```

```
typedef struct {  
    Polar p ;  
    Cartesian c ;  
    int i ;  
} structtype ;
```

```
structtype s ;
```

```
s: s.p:
```

Polar

```
s.c:
```

Cartesian

```
s.i:
```

int

Difference between Struct and Union

```
typedef union {  
    Polar p ;  
    Cartesian c ;  
    int i ;  
} uniontype ;
```

```
uniontype u ;
```

```
u: u.p:
```



```
typedef struct {  
    Polar p ;  
    Cartesian c ;  
    int i ;  
} structtype ;
```

```
structtype s ;
```

```
s: s.p:
```



```
s.c:
```

```
s.i:
```

Difference between Struct and Union

```
typedef union {  
    Polar p ;  
    Cartesian c ;  
    int i ;  
} uniontype ;
```

```
uniontype u ;
```

```
u: u.p:
```



Polar

```
typedef struct {  
    Polar p ;  
    Cartesian c ;  
    int i ;  
} structtype ;
```

```
structtype s ;
```

```
s: s.p:
```



Polar

```
s.c:
```

Cartesian

```
s.i:
```

int

Difference between Struct and Union

```
typedef union {  
    Polar p ;  
    Cartesian c ;  
    int i ;  
} uniontype ;
```

```
uniontype u ;
```

```
u: u.c:
```

Cartesian

```
typedef struct {  
    Polar p ;  
    Cartesian c ;  
    int i ;  
} structtype ;
```

```
structtype s ;
```

```
s: s.p:
```

Polar

```
s.c:
```

Cartesian

```
s.i:
```

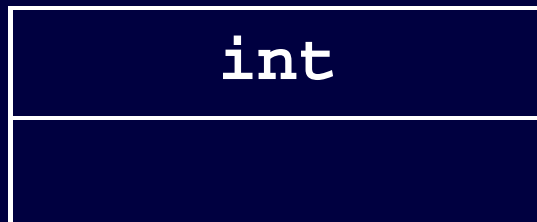
int

Difference between Struct and Union

```
typedef union {  
    Polar p ;  
    Cartesian c ;  
    int i ;  
} uniontype ;
```

```
uniontype u ;
```

```
u: u.i:
```



```
typedef struct {  
    Polar p ;  
    Cartesian c ;  
    int i ;  
} structtype ;
```

```
structtype s ;
```

```
s: s.p:
```

```
s.c:
```

```
s.i:
```



Difference between Struct and Union

```
typedef union {  
    Polar p ;  
    Cartesian c ;  
    int i ;  
} uniontype ;
```

```
uniontype u ;
```

```
u: u.p:
```



```
typedef struct {  
    Polar p ;  
    Cartesian c ;  
    int i ;  
} structtype ;
```

```
structtype s ;
```

```
s: s.p:
```



```
s.c:
```

```
s.i:
```

What is in a union?

```
typedef struct {  
  
    union {  
        Cartesian c ;  
        Polar      p ;  
    } pt ;  
} Point ;
```

There is nothing to tell you.

- C philosophy: the programmer should know what is in a union.
- Maintain an extra member, a “tag”
- The tag specifies whether the union stores a polar or a cartesian

What is in a union?

```
typedef enum { IsPolar, IsCartesian } Pointtag ;
```

```
typedef struct {  
    Pointtag tag ;  
    union {  
        Cartesian c ;  
        Polar      p ;  
    } pt ;  
} Point ;
```

There is nothing to tell you.

- C philosophy: the programmer should know what is in a union.

Creating the union

```
int main( void ) {
    Point s, t ;
    s.tag = IsCartesian ;
    s.pt.c.x = 2 ;
    s.pt.c.y = 1 ;

    t.tag = IsPolar ;
    t.pt.p.d = 2.236 ;
    t.pt.p.alpha = 0.4636 ;
}
```

By inspecting the tag, the function `rotate` can now be defined properly

Using the union

```
Point rotate( Point s, double phi ) {
    Point t ;
    if( s.tag == IsCartesian ) {
        t.tag = IsCartesian ;
        t.pt.c.x=s.pt.c.x*sin(phi)+s.pt.c.y*cos(phi);
        t.pt.c.y=s.pt.c.y*sin(phi)-s.pt.c.x*cos(phi);
    } else {
        t.tag = IsPolar ;
        t.pt.p.alpha = phi + s.pt.p.alpha ;
        t.pt.p.d = s.pt.p.d ;
    }
    return t ;
}
```

Using the union

```
Point rotate( Point s, double phi ) {
    Point t ;
    if( s.tag == IsCartesian ) {
        t.tag = IsCartesian ;
        t.pt.c.x=s.pt.c.x*sin(phi)+s.pt.c.y*cos(phi);
        t.pt.c.y=s.pt.c.y*sin(phi)-s.pt.c.x*cos(phi);
    } else {
        t.tag = IsPolar ;
        t.pt.p.alpha = phi + s.pt.p.alpha ;
        t.pt.p.d = s.pt.p.d ;
    }
    return t ;
}
```

Using the union

```
Point rotate( Point s, double phi ) {
    Point t ;
    if( s.tag == IsCartesian ) {
        t.tag = IsCartesian ;
        t.pt.c.x=s.pt.c.x*sin(phi)+s.pt.c.y*cos(phi);
        t.pt.c.y=s.pt.c.y*sin(phi)-s.pt.c.x*cos(phi);
    } else {
        t.tag = IsPolar ;
        t.pt.p.alpha = phi + s.pt.p.alpha ;
        t.pt.p.d = s.pt.p.d ;
    }
    return t ;
}
```

Using the union

```
Point rotate( Point s, double phi ) {
    Point t ;
    if( s.tag == IsCartesian ) {
        t.tag = IsCartesian ;
        t.pt.c.x=s.pt.c.x*sin(phi)+s.pt.c.y*cos(phi);
        t.pt.c.y=s.pt.c.y*sin(phi)-s.pt.c.x*cos(phi);
    } else {
        t.tag = IsPolar ;
        t.pt.p.alpha = phi + s.pt.p.alpha ;
        t.pt.p.d = s.pt.p.d ;
    }
    return t ;
}
```

Use a switch statement instead

```
Point rotate( Point s, double phi ) {
    Point t ;
    switch( s.tag ) {
        case IsCartesian:
            t.tag = IsCartesian ;
            t.pt.c.x=s.pt.c.x*sin(phi)+s.pt.c.y*cos(phi);
            t.pt.c.y=s.pt.c.y*sin(phi)-s.pt.c.x*cos(phi);
            break;
        case IsPolar:
            t.tag = IsPolar ;
            t.pt.p.alpha = phi + s.pt.p.alpha ;
            t.pt.p.d = s.pt.p.d ;
            break;
    }
    return t ;
}
```

Use a switch statement instead

```
Point rotate( Point s, double phi ) {  
    Point t ;  
    switch( s.tag ) {  
        case IsCartesian:  
            t.tag = IsCartesian ;  
            t.pt.c.x=s.pt.c.x*sin(phi)+s.pt.c.y*cos(phi) ;  
            t.pt.c.y=s.pt.c.y*sin(phi)-s.pt.c.x*cos(phi) ;  
            break ;  
        case IsPolar:  
            t.tag = IsPolar ;  
            t.pt.p.alpha = phi + s.pt.p.alpha ;  
            t.pt.p.d = s.pt.p.d ;  
            break ;  
    }  
    return t ;  
}
```

Use a switch statement instead

```
Point rotate( Point s, double phi ) {
    Point t ;
    switch( s.tag ) {
        case IsCartesian:
            t.tag = IsCartesian ;
            t.pt.c.x=s.pt.c.x*sin(phi)+s.pt.c.y*cos(phi) ;
            t.pt.c.y=s.pt.c.y*sin(phi)-s.pt.c.x*cos(phi) ;
            break;
        case IsPolar:
            t.tag = IsPolar ;
            t.pt.p.alpha = phi + s.pt.p.alpha ;
            t.pt.p.d = s.pt.p.d ;
            break;
    }
    return t ;
}
```

Use a switch statement instead

```
Point rotate( Point s, double phi ) {
    Point t ;
    switch( s.tag ) {
        case IsCartesian:
            t.tag = IsCartesian ;
            t.pt.c.x=s.pt.c.x*sin(phi)+s.pt.c.y*cos(phi);
            t.pt.c.y=s.pt.c.y*sin(phi)-s.pt.c.x*cos(phi);
            break;
        case IsPolar:
            t.tag = IsPolar ;
            t.pt.p.alpha = phi + s.pt.p.alpha ;
            t.pt.p.d = s.pt.p.d ;
            break;
    }
    return t ;
}
```

Summarising Flat Data

Constructs:

```
typedef enum { value1, value2, ... } typename ;  
typedef struct { type1 x1 ; type2 x2 ... } typename  
typedef union { type1 x1 ; type2 x2 ... } typename  
switch( value ) { case 1: ... ; break ; case 2: ...
```

Coming lectures:

- Dynamic memory
- (• more on pointers!)
- Non flat data.

Coming lectures: non flat data

- Dynamic memory
 - `ptr = calloc(num, size)`
 - `free(ptr)`
- Dynamic data structures
 - Lists

Where is data stored

- Data can be stored in three places.
 - Local variables / function parameters
 - * Stored on the stack
 - * Each function invocation has its own set of locals
 - * Allocated when the function starts; De-allocated when the function ends.
 - *Global variables*
 - * Stored in the data segment
 - * There is only one data segment shared by all functions
 - * Allocated when the program starts; De-allocated when the program ends.
 - The *heap*
 - * Shared by all functions
 - * Allocated by the programmer

Global variables

Global variables are different from Local variables in two senses:

1. Their lifetime is longer

- Global variables keep their value throughout the execution of the program

2. Their scope is larger

- Global variables can be seen and used by all functions.

Example: good use of global

```
int dice_state ;

int dice( void ) {
    dice_state = ( dice_state * 21 + 1 ) % 1024 ;
    return dice_state % 6 + 1 ;
}

int main( void ) {
    printf( "Dice value: %d\n", dice() ) ;
    printf( "Dice value: %d\n", dice() ) ;
    printf( "Dice value: %d\n", dice() ) ;
    printf( "Dice value: %d\n", dice() ) ;
    printf( "Dice value: %d\n", dice() ) ;
}
Will print 2, 5, 2, 5, 4
```

Example: BAD use of global

```
int z ;
```

```
void dosomething( void ) {  
    z = 4 ;  
}
```

```
int main( void ) {  
    z = 5 ;  
    dosomething() ;  
    printf( "Z value: %d\n", z ) ;  
}
```

Will print 4. Not intuitive from `main`.

Even worse, you can have a global and a local with the same name.
Now that will throw everyone.

Dynamic data, the tool

The tool for dynamic data is `calloc`

- The function `calloc` allocates some memory cells and returns a pointer to these cells.
- `calloc(num, sizeof(type))` allocates sufficient space to hold `num` elements of `type`

For example

```
#include<stdlib.h>
```

```
...
```

```
int *intarray = calloc( 10, sizeof( int ) ) ;
```

```
double *darray = calloc( 100, sizeof( double ) ) ;
```

will allocate space for an array of 10 integers, and an array of 100 doubles.

Calloced data

Data allocated with `calloc` will stay alive as long as you want, until you call `free`.

```
int i ;
int *intarray = calloc( 10, sizeof( int ) ) ;
for( i=0 ; i<10 ; i++ ) {
    intarray[i] = i*i ;
}
for( i=0 ; i<10 ; i++ ) {
    printf( "%d: %d\n", i, intarray[i] ) ;
}
free( intarray ) ;
```

Return type of calloc

Calloc will return a pointer to something of the appropriate type. If memory is available, this pointer is a pointer to the memory cell. If no memory is available, this pointer is a “NULL” pointer.

- NULL is not a real pointer
- You cannot dereference NULL
- NULL is used to denote special pointer values
 - Error conditions (no more memory)
 - End of lists (as we will see in a second)

You ought to check every time you call calloc whether the result is NULL or not. If it is NULL, you must take evasive action...

Clever uses of calloc - I

Make a copy of a string that lives forever:

```
#include <stdlib.h>
```

```
char *strdup( char *s ) {  
    char *result = calloc( strlen( s ) + 1,  
                           sizeof( char ) ) ;  
    if( result == NULL ) {  
        fprintf( stderr, "HELP, out of memory!\n" ) ;  
    } else {  
        strcpy( result, s ) ;  
    }  
    return result ;  
}
```

Clever uses of calloc - II

Concatenate two strings:

```
#include <stdlib.h>

char *concat( char *head, char *tail ) {
    char *result = calloc( strlen( head ) +
                          strlen( tail ) + 1,
                          sizeof( char ) ) ;
    /* CHECK ON NULL! */
    strcpy( result, head ) ;
    strcat( result, tail ) ;
    return result ;
}
```

Clever uses of calloc - III

Design a phonebook.

```
typedef struct {
    char name[ 20 ] ;
    char number[ 12 ] ;
} phonebook ;

phonebook *newphonebook( ) {
    phonebook *r = calloc( 50000000,
                          sizeof( phonebook ) ) ;
    strcpy( r[0].name, "Aardvark" ) ;
    strcpy( r[0].number, "0117-9093145" ) ;
    return r ;
}
```

Phonebook?

Right, suppose that we had our phonebook, and we wanted to add one name...

- Must allocate the whole lot again.
- Must copy the whole lot over in the newly allocated space.
- Not too pretty.

Solution:

- Use a dynamic data structure.

Dynamic versus static structures

Static data structures store a fixed number of items

- Useful to store a date, or a salary, or a personnel record.
- But, how to store our phonebook?
- 50000000 entries?
- 50000001 entries?

Non flat data structures cater for this, and much more:

- A List of numbers is a group (number, List of numbers)

`(3, (4, (5, (6, ...))))`

`(1, (4, (9, (16, ...))))`

Dynamic versus static structures

Static data structures store a fixed number of items

- Useful to store a date, or a salary, or a personnel record.
- But, how to store our phonebook?
- 50000000 entries?
- 50000001 entries?

Non flat data structures cater for this, and much more:

- A List of numbers is a group (number, List of numbers)

$(3, (4, (5, (6, \dots))))$

$(1, (4, (9, (16, \dots))))$

Dynamic versus static structures

Static data structures store a fixed number of items

- Useful to store a date, or a salary, or a personnel record.
- But, how to store our phonebook?
- 50000000 entries?
- 50000001 entries?

Non flat data structures cater for this, and much more:

- A List of numbers is a group (number, List of numbers)

`(3, (4, (5, (6, ...))))`

`(1, (4, (9, (16, ...))))`

Dynamic versus static structures

Static data structures store a fixed number of items

- Useful to store a date, or a salary, or a personnel record.
- But, how to store our phonebook?
- 50000000 entries?
- 50000001 entries?

Non flat data structures cater for this, and much more:

- A List of numbers is a group (number, List of numbers)

`(3, (4, (5, (6, ...))))`

`(1, (4, (9, (16, ...))))`

Dynamic versus static structures

Static data structures store a fixed number of items

- Useful to store a date, or a salary, or a personnel record.
- But, how to store our phonebook?
- 50000000 entries?
- 50000001 entries?

Non flat data structures cater for this, and much more:

- A **List of numbers** is a group (number, **List of numbers**)

$(3, (4, (5, (6, \dots))))$

$(1, (4, (9, (16, \dots))))$

Non-flat: A data structure defined in terms of itself.

⇒ A recursive data structure

Building data structures

What we need is an element in the list, and a *pointer* to the rest of the list

```
typedef struct xxlistxx {  
    int n ;  
    struct xxlistxx *next ;  
} list ;
```

- Note the syntax

Building data structures

What we need is an element in the list, and a *pointer* to the rest of the list

```
typedef struct xxlistxx {  
    int n ;  
    struct xxlistxx *next ;  
} list ;
```

- This is the structure tag, together with the word `struct`

Building data structures

What we need is an element in the list, and a *pointer* to the rest of the list

```
typedef struct xxlistxx {  
    int n ;  
    list *next ;      /* ILLEGAL, forward ref */  
} list ;
```

- This is illegal, the type is not defined yet...

Building data structures

What we need is an element in the list, and a *pointer* to the rest of the list

```
typedef struct xxlistxx {  
    int n ;  
    struct xxlistxx *next ;  
} list ;
```

- But the tag has been defined already...

How are we going to create structures?

Silly example of a list

```
typedef struct list {  
    int n ;  
    struct list *next ;  
} list ;
```

```
int main( void ) {  
    list k = {3, NULL};  
    list l = {2, &k}; /* (*(x)).a is unreadable */  
    list m = {1, &l};  
    printf( "first %d\n", m.n ) ;  
    printf( "second %d\n", (*(m.next)).n ) ;  
    printf( "third %d\n", ((*(*m.next)).next).n ) ;  
}
```

Silly example of a list

```
typedef struct list {
    int n ;
    struct list *next ;
} list ;
```

```
int main( void ) {
    list k = {3, NULL};
    list l = {2, &k}; /* (*(x)).a is unreadable */
    list m = {1, &l}; /* (x)->a is a shortcut */
    printf( "first %d\n", m.n ) ;
    printf( "second %d\n",      m.next-> n ) ;
    printf( "third %d\n",      m.next-> next-> n ) ;
}
```

Example of a list

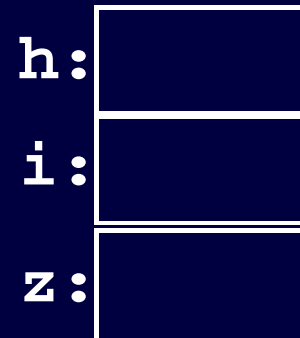
```
#include <stdlib.h>
typedef struct L { int x ; struct L *next ; } List;
List *insertList( int hd, List *tl ) {
    List *t = calloc( 1, sizeof( List ) ) ;
    t->x = hd ; t->next = tl ; return t ;
}
List *somelist( void ) {
    List *h = NULL ; int i ;
    for( i=1 ; i<5 ; i++ ) {
        h=insertList( i, h );}
    return h ;
}
int main( void ) {
    List *z = somelist() ;
}
```

Example of a list

```
#include <stdlib.h>
typedef struct L { int x ; struct L *next ; } List;
List *insertList( int hd, List *tl ) {
    List *t = calloc( 1, sizeof( List ) ) ;
    t->x = hd ; t->next = tl ; return t ;
}
List *somelist( void ) {
    List *h = NULL ; int i ;
    for( i=1 ; i<5 ; i++ ) {
        h=insertList( i, h );}
    return h ;
}
int main( void ) {
⇒ List *z = somelist() ;      z: 
}
```

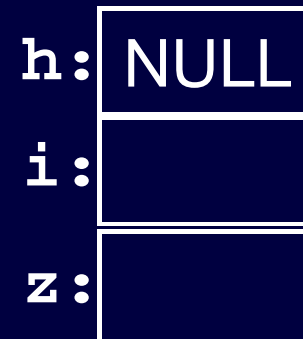
Example of a list

```
#include <stdlib.h>
typedef struct L { int x ; struct L *next ; } List;
List *insertList( int hd, List *tl ) {
    List *t = calloc( 1, sizeof( List ) ) ;
    t->x = hd ; t->next = tl ; return t ;
}
List *somelist( void ) {
⇒ List *h = NULL ; int i ;
    for( i=1 ; i<5 ; i++ ) {
        h=insertList( i, h );}
    return h ;
}
int main( void ) {
    List *z = somelist() ;
}
```



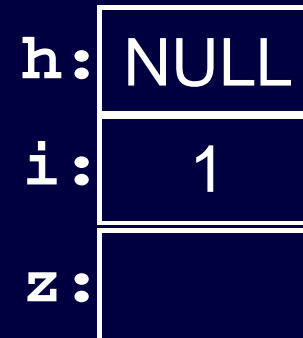
Example of a list

```
#include <stdlib.h>
typedef struct L { int x ; struct L *next ; } List;
List *insertList( int hd, List *tl ) {
    List *t = calloc( 1, sizeof( List ) ) ;
    t->x = hd ; t->next = tl ; return t ;
}
List *somelist( void ) {
    List *h = NULL ; int i ;
⇒ for( i=1 ; i<5 ; i++ ) {
    h=insertList( i, h );}
    return h ;
}
int main( void ) {
    List *z = somelist() ;
}
```



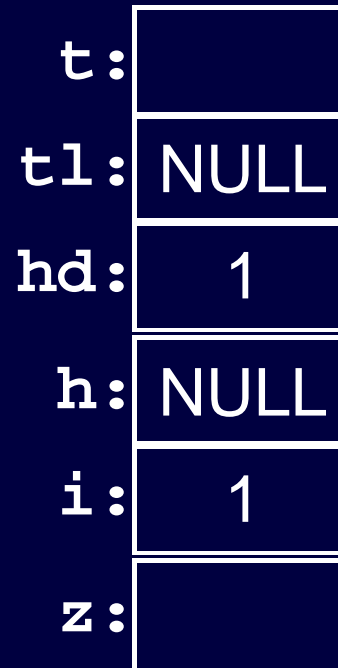
Example of a list

```
#include <stdlib.h>
typedef struct L { int x ; struct L *next ; } List;
List *insertList( int hd, List *tl ) {
    List *t = calloc( 1, sizeof( List ) ) ;
    t->x = hd ; t->next = tl ; return t ;
}
List *somelist( void ) {
    List *h = NULL ; int i ;
    for( i=1 ; i<5 ; i++ ) {
⇒     h=insertList( i, h );}
    return h ;
}
int main( void ) {
    List *z = somelist() ;
}
```



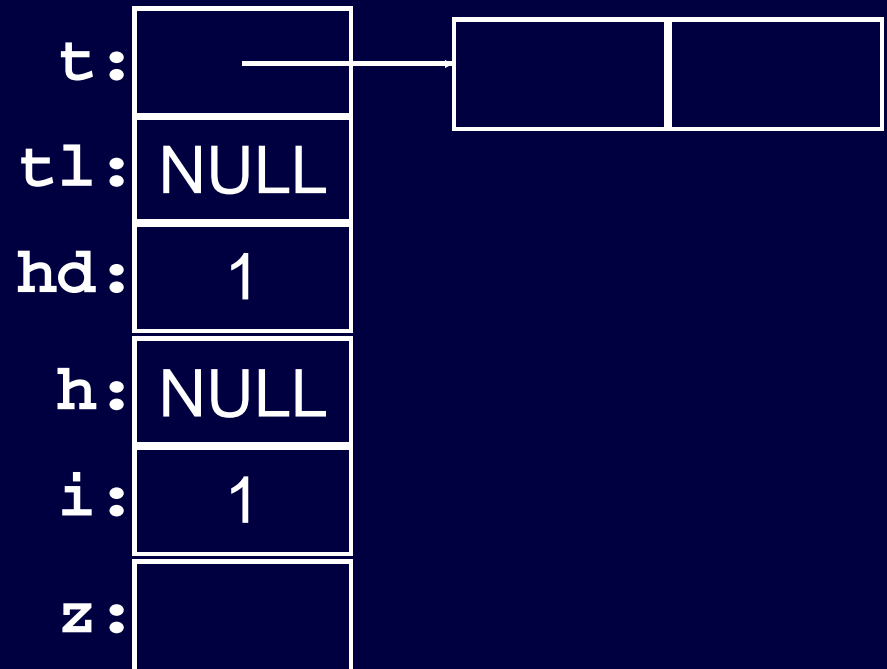
Example of a list

```
#include <stdlib.h>
typedef struct L { int x ; struct L *next ; } List;
List *insertList( int hd, List *tl ) {
⇒ List *t = calloc( 1, sizeof( List ) ) ;
  t->x = hd ; t->next = tl ; return t ;
}
List *somelist( void ) {
  List *h = NULL ; int i ;
  for( i=1 ; i<5 ; i++ ) {
    h=insertList( i, h );}
  return h ;
}
int main( void ) {
  List *z = somelist() ;
}
```



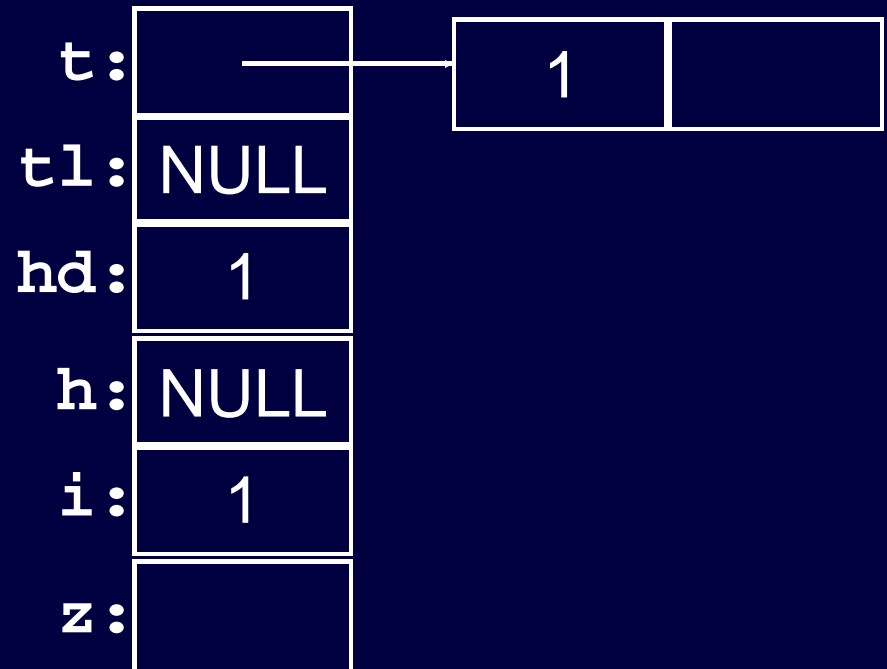
Example of a list

```
#include <stdlib.h>
typedef struct L { int x ; struct L *next ; } List;
List *insertList( int hd, List *t1 ) {
    List *t = calloc( 1, sizeof( List ) ) ;
    ⇒ t->x = hd ; t->next = t1 ; return t ;
}
List *somalist( void ) {
    List *h = NULL ; int i ;
    for( i=1 ; i<5 ; i++ ) {
        h=insertList( i, h );}
    return h ;
}
int main( void ) {
    List *z = somalist() ;
}
```



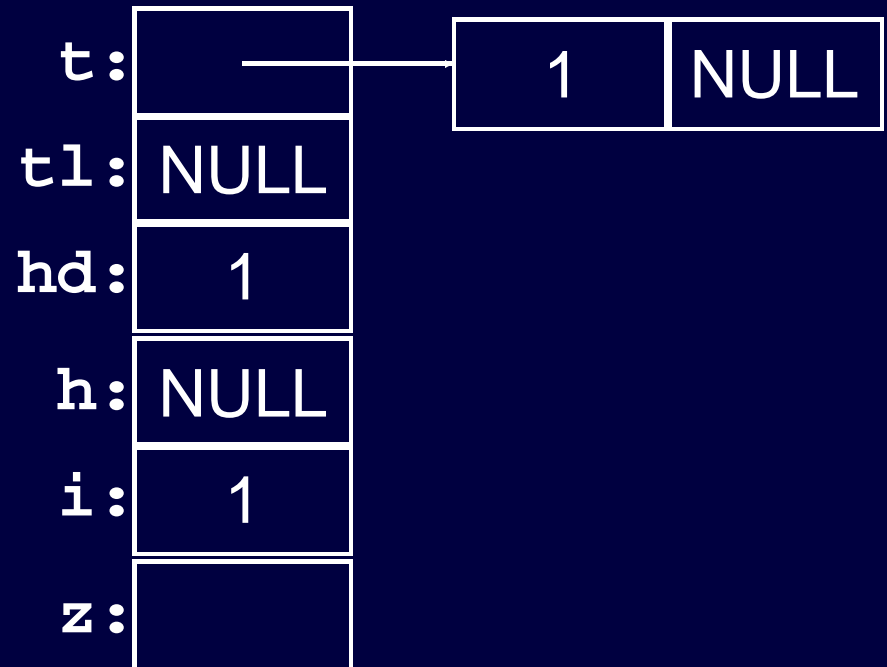
Example of a list

```
#include <stdlib.h>
typedef struct L { int x ; struct L *next ; } List;
List *insertList( int hd, List *t1 ) {
    List *t = calloc( 1, sizeof( List ) ) ;
    t->x = hd ; t->next = t1 ; return t ;
}
List *somelist( void ) {
    List *h = NULL ; int i ;
    for( i=1 ; i<5 ; i++ ) {
        h=insertList( i, h );}
    return h ;
}
int main( void ) {
    List *z = somelist() ;
}
```



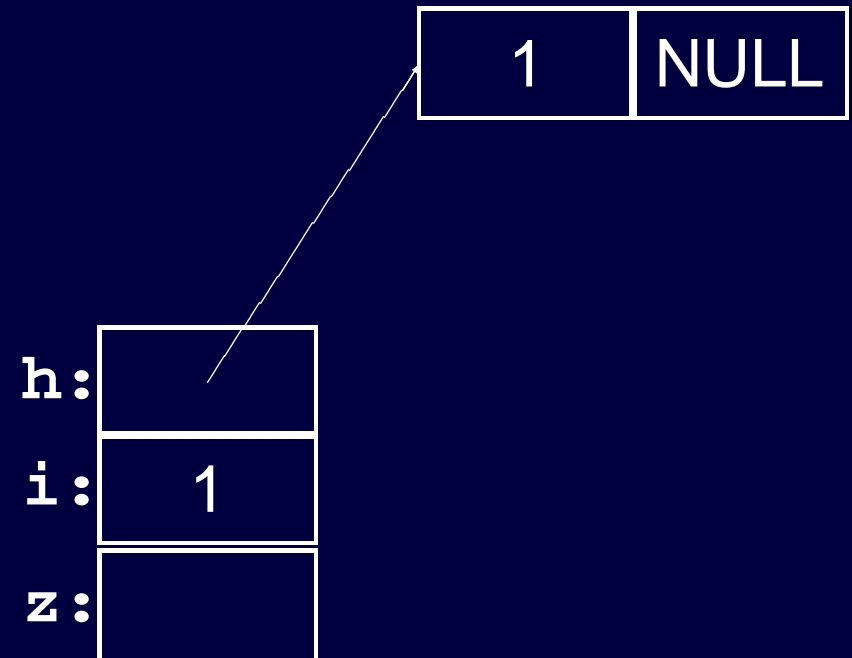
Example of a list

```
#include <stdlib.h>
typedef struct L { int x ; struct L *next ; } List;
List *insertList( int hd, List *t1 ) {
    List *t = calloc( 1, sizeof( List ) ) ;
    t->x = hd ; t->next = t1 ; return t ;
}
List *somelist( void ) {
    List *h = NULL ; int i ;
    for( i=1 ; i<5 ; i++ ) {
        h=insertList( i, h );}
    return h ;
}
int main( void ) {
    List *z = somelist() ;
}
```



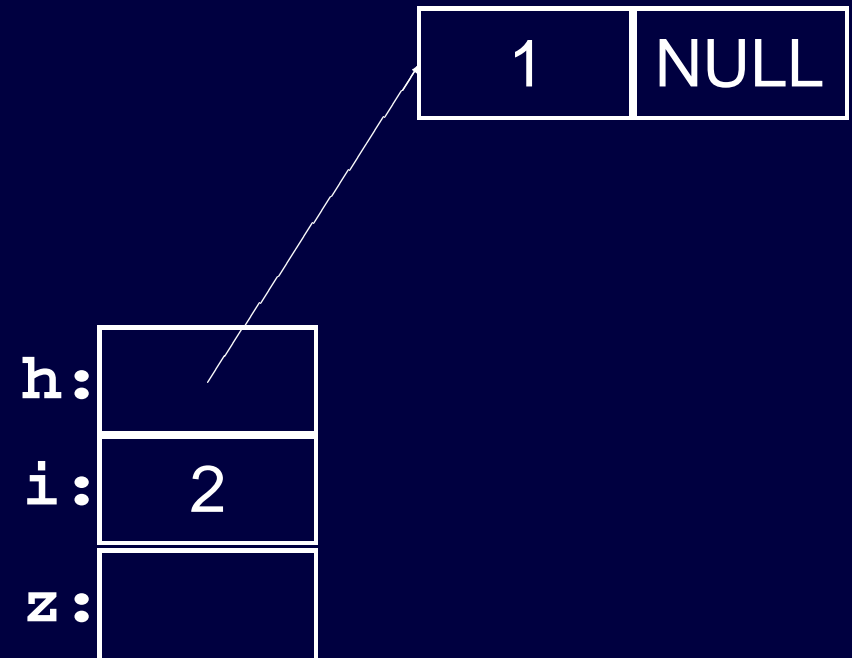
Example of a list

```
#include <stdlib.h>
typedef struct L { int x ; struct L *next ; } List;
List *insertList( int hd, List *tl ) {
    List *t = calloc( 1, sizeof( List ) ) ;
    t->x = hd ; t->next = tl ; return t ;
}
List *somelist( void ) {
    List *h = NULL ; int i ;
⇒ for( i=1 ; i<5 ; i++ ) {
    h=insertList( i, h );}
    return h ;
}
int main( void ) {
    List *z = somelist() ;
}
```



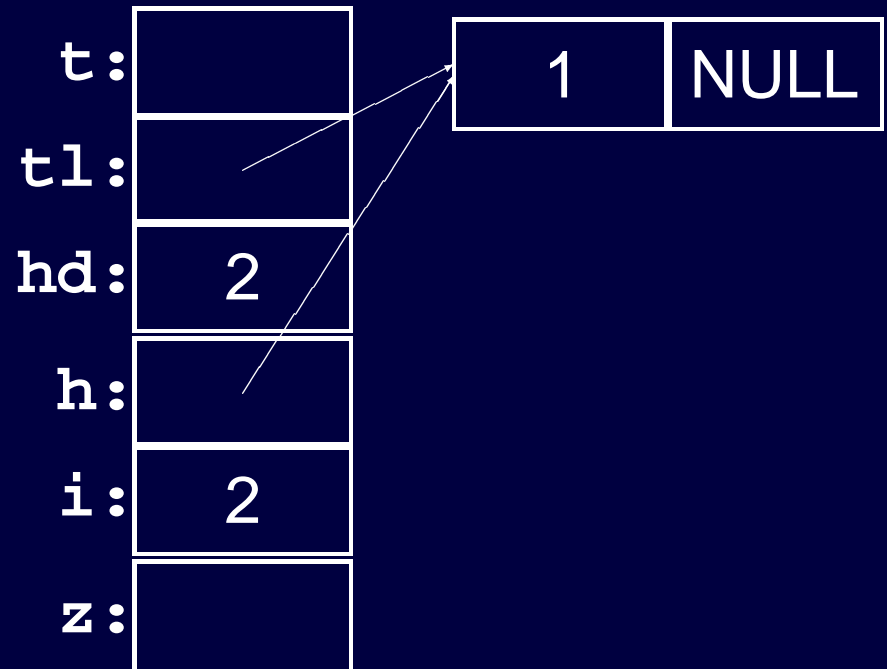
Example of a list

```
#include <stdlib.h>
typedef struct L { int x ; struct L *next ; } List;
List *insertList( int hd, List *tl ) {
    List *t = calloc( 1, sizeof( List ) ) ;
    t->x = hd ; t->next = tl ; return t ;
}
List *somelist( void ) {
    List *h = NULL ; int i ;
    for( i=1 ; i<5 ; i++ ) {
⇒    h=insertList( i, h );}
    return h ;
}
int main( void ) {
    List *z = somelist() ;
}
```



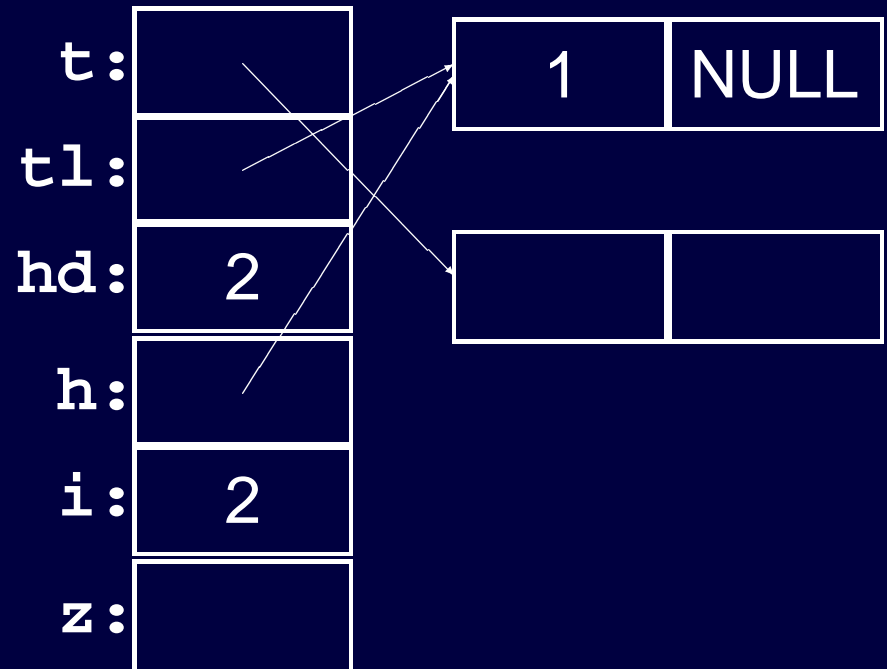
Example of a list

```
#include <stdlib.h>
typedef struct L { int x ; struct L *next ; } List;
List *insertList( int hd, List *tl ) {
⇒ List *t = calloc( 1, sizeof( List ) ) ;
  t->x = hd ; t->next = tl ; return t ;
}
List *somelist( void ) {
  List *h = NULL ; int i ;
  for( i=1 ; i<5 ; i++ ) {
    h=insertList( i, h );}
  return h ;
}
int main( void ) {
  List *z = somelist() ;
}
```



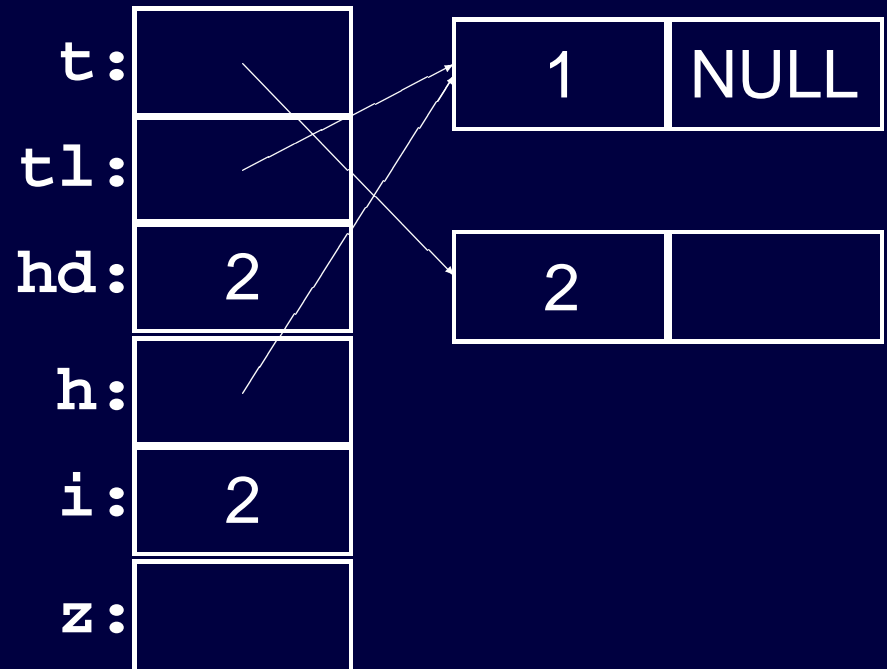
Example of a list

```
#include <stdlib.h>
typedef struct L { int x ; struct L *next ; } List;
List *insertList( int hd, List *tl ) {
    List *t = calloc( 1, sizeof( List ) ) ;
    ⇒ t->x = hd ; t->next = tl ; return t ;
}
List *somelist( void ) {
    List *h = NULL ; int i ;
    for( i=1 ; i<5 ; i++ ) {
        h=insertList( i, h );}
    return h ;
}
int main( void ) {
    List *z = somelist() ;
}
```



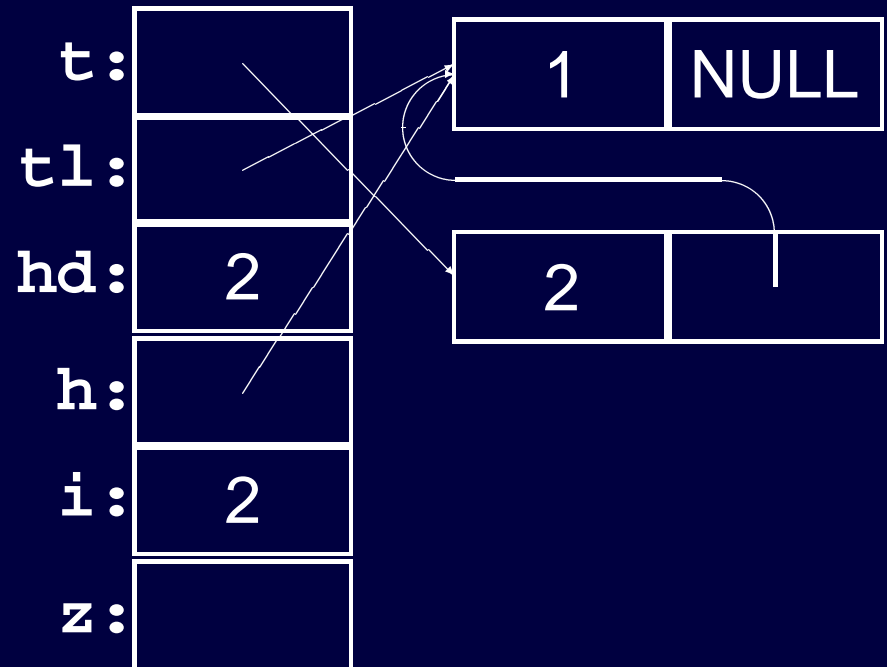
Example of a list

```
#include <stdlib.h>
typedef struct L { int x ; struct L *next ; } List;
List *insertList( int hd, List *tl ) {
    List *t = calloc( 1, sizeof( List ) ) ;
    t->x = hd ; t->next = tl ; return t ;
}
List *somelist( void ) {
    List *h = NULL ; int i ;
    for( i=1 ; i<5 ; i++ ) {
        h=insertList( i, h );}
    return h ;
}
int main( void ) {
    List *z = somelist() ;
}
```



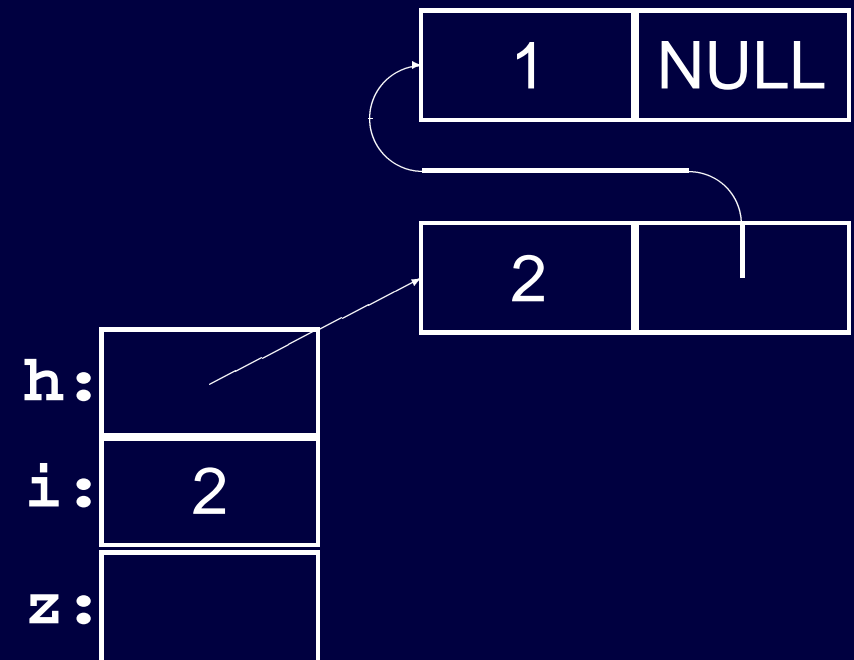
Example of a list

```
#include <stdlib.h>
typedef struct L { int x ; struct L *next ; } List;
List *insertList( int hd, List *tl ) {
    List *t = calloc( 1, sizeof( List ) ) ;
    t->x = hd ; t->next = tl ; return t ;
}
List *somelist( void ) {
    List *h = NULL ; int i ;
    for( i=1 ; i<5 ; i++ ) {
        h=insertList( i, h );}
    return h ;
}
int main( void ) {
    List *z = somelist() ;
}
```



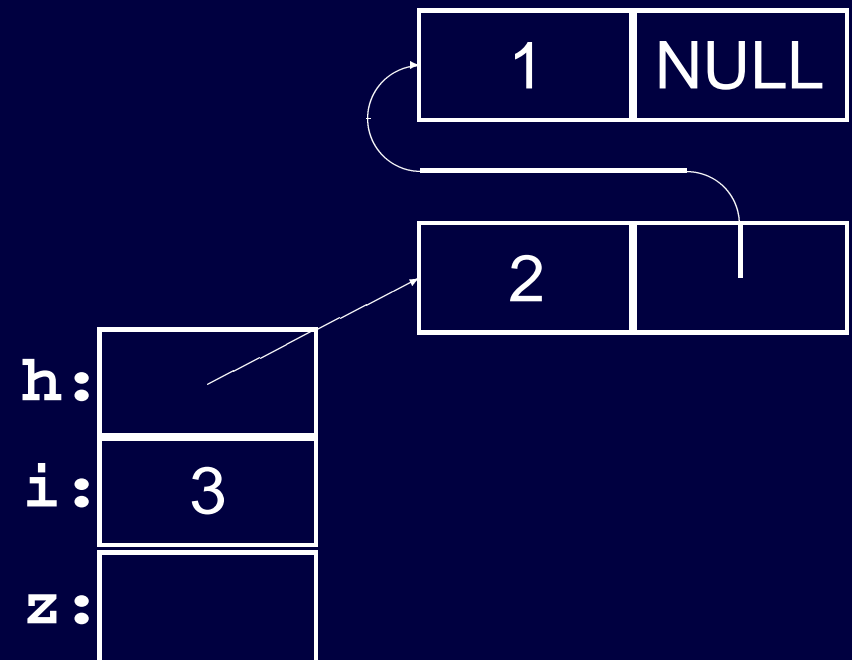
Example of a list

```
#include <stdlib.h>
typedef struct L { int x ; struct L *next ; } List;
List *insertList( int hd, List *tl ) {
    List *t = calloc( 1, sizeof( List ) ) ;
    t->x = hd ; t->next = tl ; return t ;
}
List *somelist( void ) {
    List *h = NULL ; int i ;
    ⇒ for( i=1 ; i<5 ; i++ ) {
        h=insertList( i, h );}
    return h ;
}
int main( void ) {
    List *z = somelist() ;
}
```



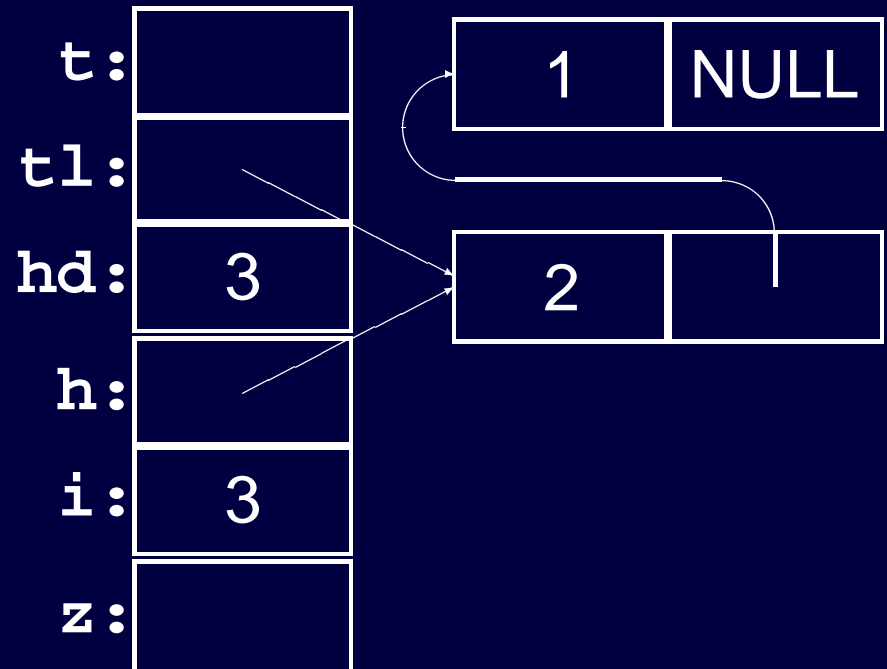
Example of a list

```
#include <stdlib.h>
typedef struct L { int x ; struct L *next ; } List;
List *insertList( int hd, List *tl ) {
    List *t = calloc( 1, sizeof( List ) ) ;
    t->x = hd ; t->next = tl ; return t ;
}
List *somelist( void ) {
    List *h = NULL ; int i ;
    for( i=1 ; i<5 ; i++ ) {
⇒     h=insertList( i, h );}
    return h ;
}
int main( void ) {
    List *z = somelist() ;
}
```



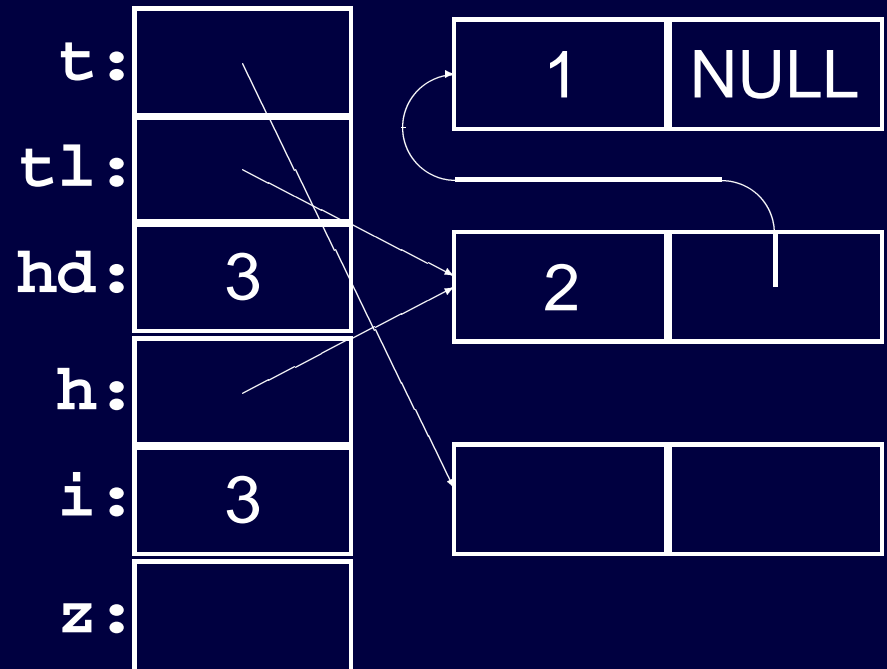
Example of a list

```
#include <stdlib.h>
typedef struct L { int x ; struct L *next ; } List;
List *insertList( int hd, List *tl ) {
⇒ List *t = calloc( 1, sizeof( List ) ) ;
  t->x = hd ; t->next = tl ; return t ;
}
List *somalist( void ) {
  List *h = NULL ; int i ;
  for( i=1 ; i<5 ; i++ ) {
    h=insertList( i, h );}
  return h ;
}
int main( void ) {
  List *z =omalist() ;
}
```



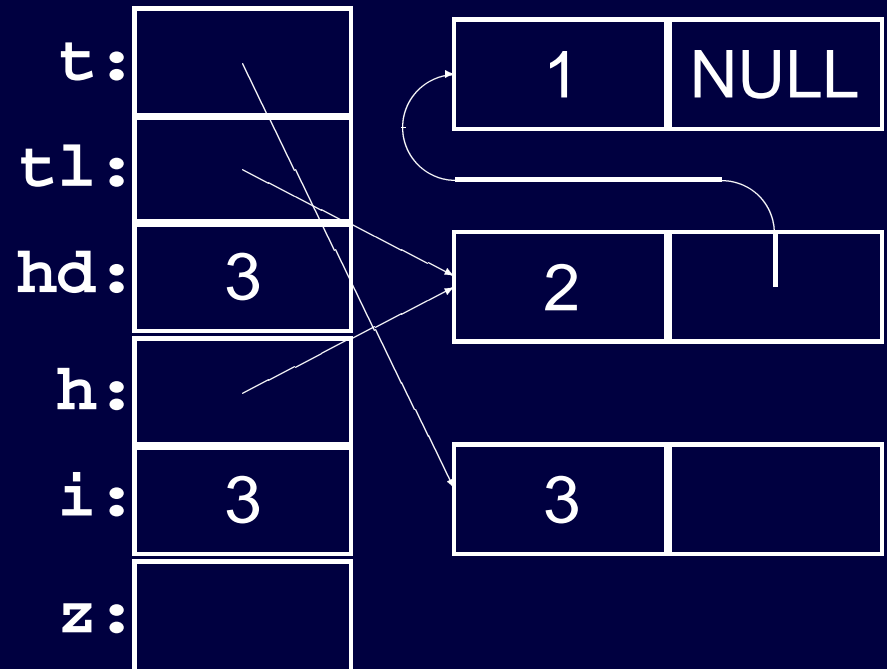
Example of a list

```
#include <stdlib.h>
typedef struct L { int x ; struct L *next ; } List;
List *insertList( int hd, List *tl ) {
    List *t = calloc( 1, sizeof( List ) ) ;
    ⇒ t->x = hd ; t->next = tl ; return t ;
}
List *somelist( void ) {
    List *h = NULL ; int i ;
    for( i=1 ; i<5 ; i++ ) {
        h=insertList( i, h );}
    return h ;
}
int main( void ) {
    List *z = somelist() ;
}
```



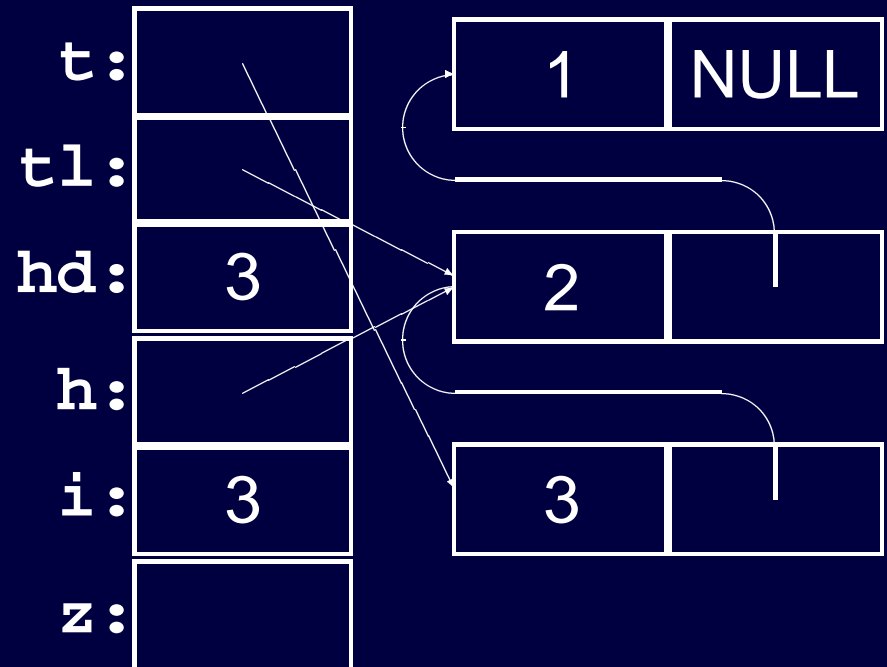
Example of a list

```
#include <stdlib.h>
typedef struct L { int x ; struct L *next ; } List;
List *insertList( int hd, List *tl ) {
    List *t = calloc( 1, sizeof( List ) ) ;
    t->x = hd ; t->next = tl ; return t ;
}
List *somelist( void ) {
    List *h = NULL ; int i ;
    for( i=1 ; i<5 ; i++ ) {
        h=insertList( i, h );}
    return h ;
}
int main( void ) {
    List *z = somelist() ;
}
```



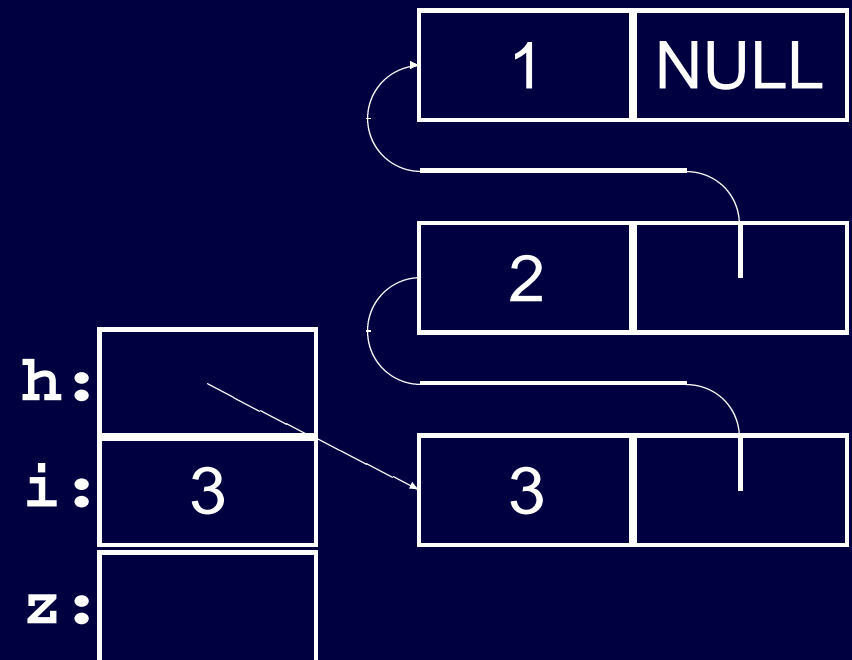
Example of a list

```
#include <stdlib.h>
typedef struct L { int x ; struct L *next ; } List;
List *insertList( int hd, List *tl ) {
    List *t = calloc( 1, sizeof( List ) ) ;
    t->x = hd ; t->next = tl ; return t ;
}
List *somelist( void ) {
    List *h = NULL ; int i ;
    for( i=1 ; i<5 ; i++ ) {
        h=insertList( i, h );}
    return h ;
}
int main( void ) {
    List *z = somelist() ;
}
```



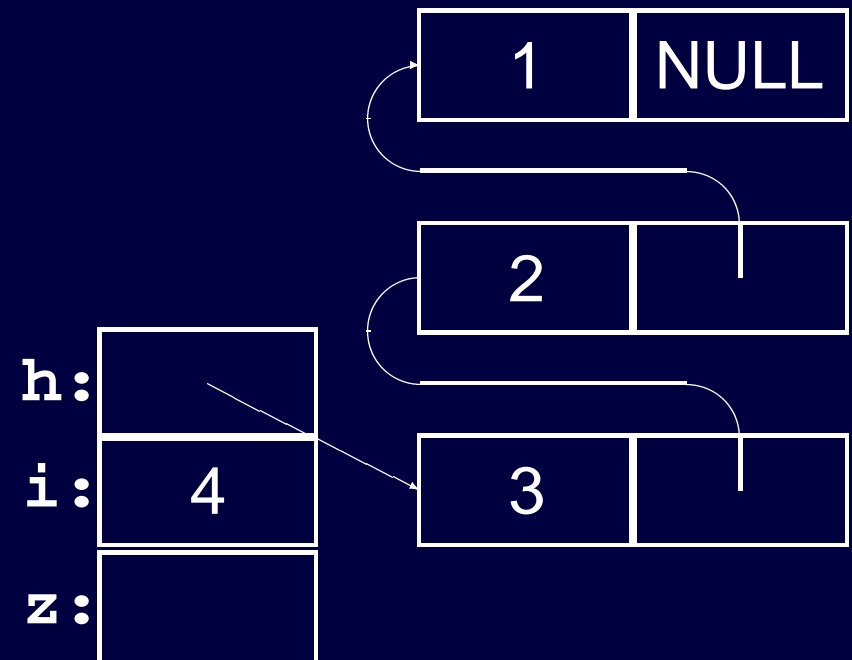
Example of a list

```
#include <stdlib.h>
typedef struct L { int x ; struct L *next ; } List;
List *insertList( int hd, List *tl ) {
    List *t = calloc( 1, sizeof( List ) ) ;
    t->x = hd ; t->next = tl ; return t ;
}
List *somelist( void ) {
    List *h = NULL ; int i ;
    ⇒ for( i=1 ; i<5 ; i++ ) {
        h=insertList( i, h );}
    return h ;
}
int main( void ) {
    List *z = somelist() ;
}
```



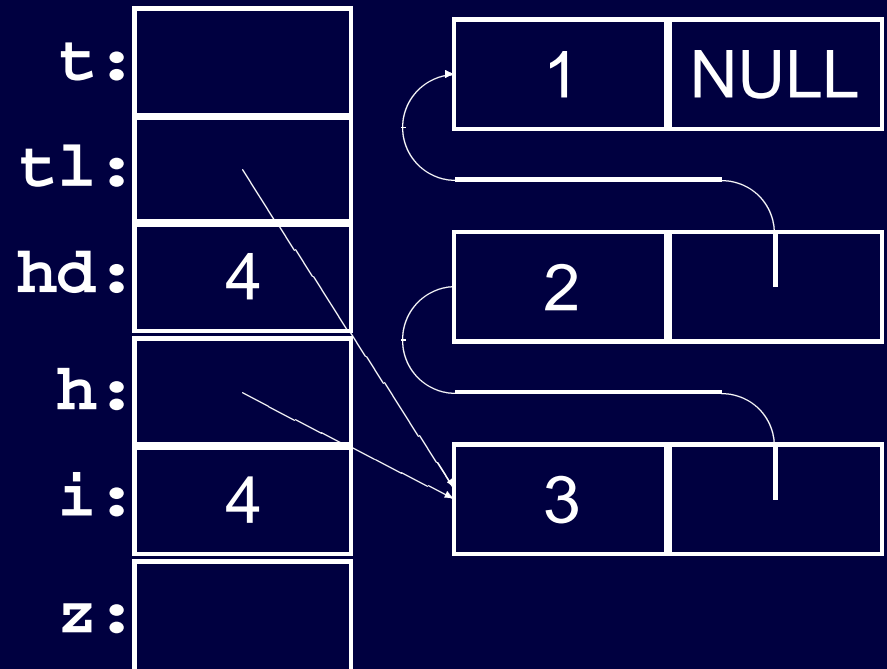
Example of a list

```
#include <stdlib.h>
typedef struct L { int x ; struct L *next ; } List;
List *insertList( int hd, List *tl ) {
    List *t = calloc( 1, sizeof( List ) ) ;
    t->x = hd ; t->next = tl ; return t ;
}
List *somelist( void ) {
    List *h = NULL ; int i ;
    for( i=1 ; i<5 ; i++ ) {
⇒     h=insertList( i, h );}
    return h ;
}
int main( void ) {
    List *z = somelist() ;
}
```



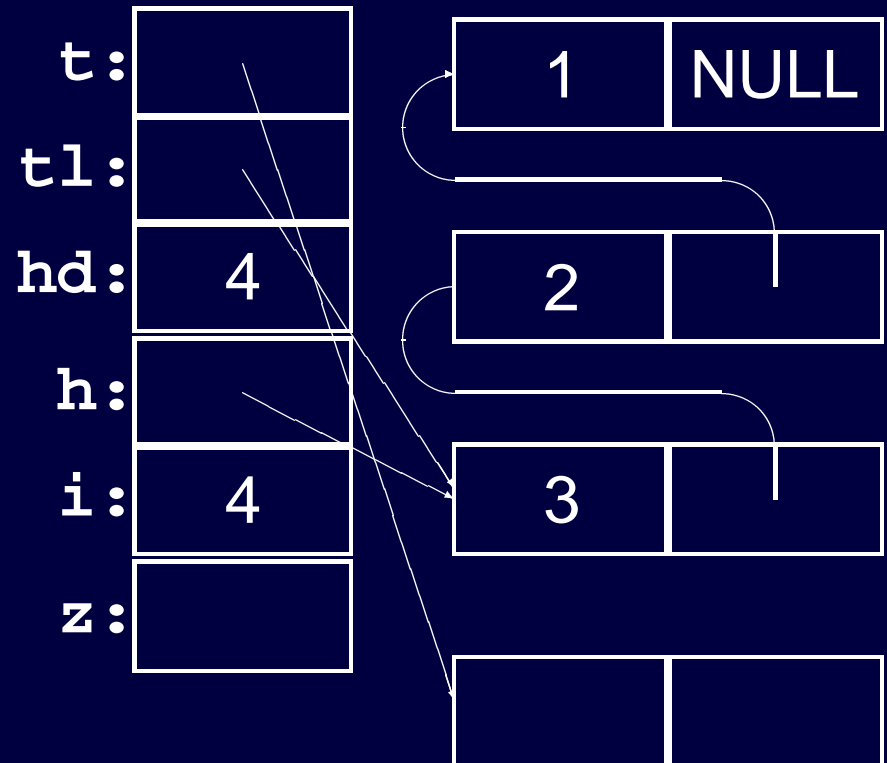
Example of a list

```
#include <stdlib.h>
typedef struct L { int x ; struct L *next ; } List;
List *insertList( int hd, List *tl ) {
⇒ List *t = calloc( 1, sizeof( List ) ) ;
  t->x = hd ; t->next = tl ; return t ;
}
List *somelist( void ) {
  List *h = NULL ; int i ;
  for( i=1 ; i<5 ; i++ ) {
    h=insertList( i, h );}
  return h ;
}
int main( void ) {
  List *z = somelist() ;
}
```



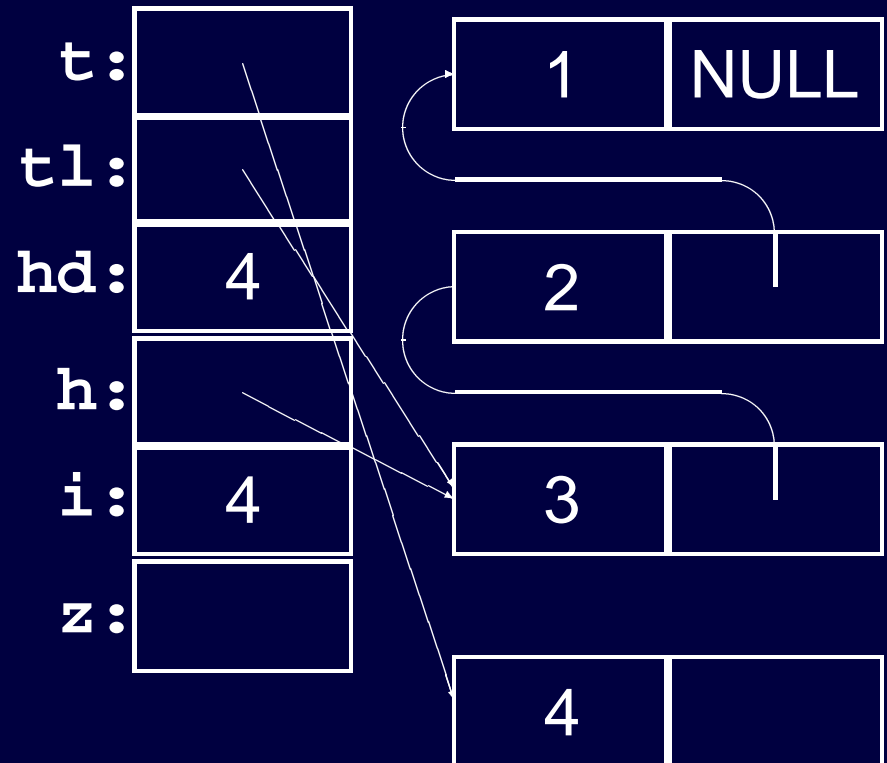
Example of a list

```
#include <stdlib.h>
typedef struct L { int x ; struct L *next ; } List;
List *insertList( int hd, List *tl ) {
    List *t = calloc( 1, sizeof( List ) ) ;
    ⇒ t->x = hd ; t->next = tl ; return t ;
}
List *somalist( void ) {
    List *h = NULL ; int i ;
    for( i=1 ; i<5 ; i++ ) {
        h=insertList( i, h );}
    return h ;
}
int main( void ) {
    List *z = somalist() ;
}
```



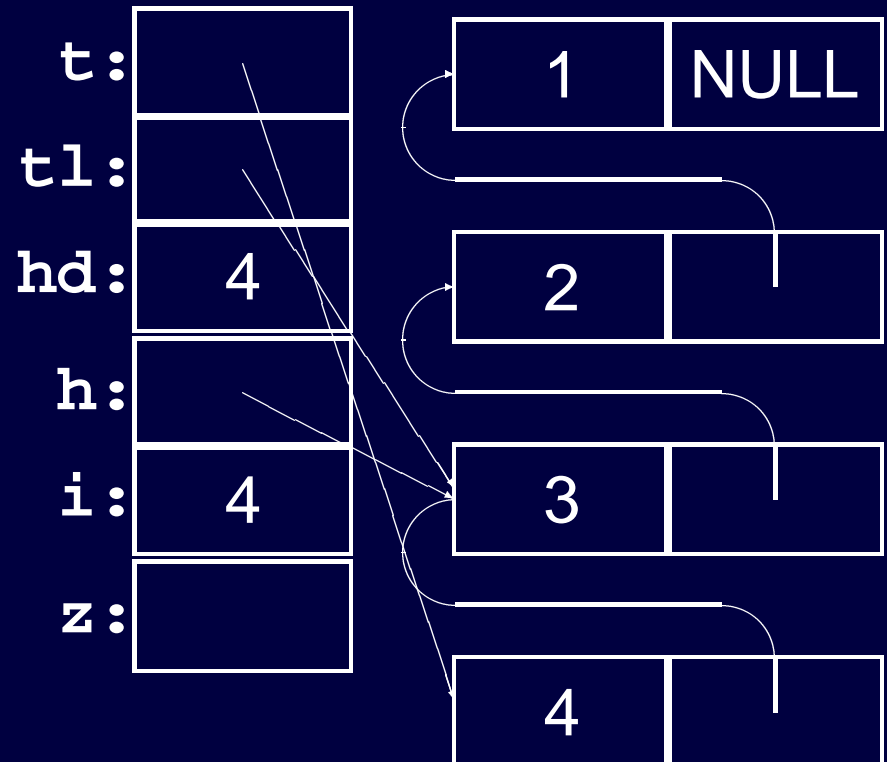
Example of a list

```
#include <stdlib.h>
typedef struct L { int x ; struct L *next ; } List;
List *insertList( int hd, List *tl ) {
    List *t = calloc( 1, sizeof( List ) ) ;
    t->x = hd ; t->next = tl ; return t ;
}
List *somelist( void ) {
    List *h = NULL ; int i ;
    for( i=1 ; i<5 ; i++ ) {
        h=insertList( i, h );}
    return h ;
}
int main( void ) {
    List *z = somelist() ;
}
```



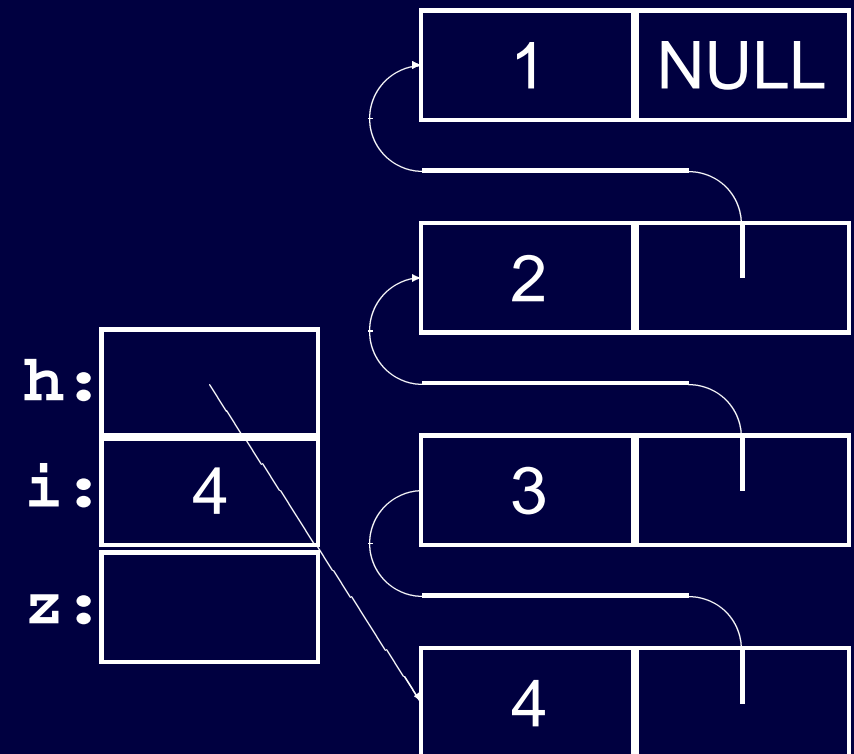
Example of a list

```
#include <stdlib.h>
typedef struct L { int x ; struct L *next ; } List;
List *insertList( int hd, List *tl ) {
    List *t = calloc( 1, sizeof( List ) ) ;
    t->x = hd ; t->next = tl ; return t ;
}
List *somelist( void ) {
    List *h = NULL ; int i ;
    for( i=1 ; i<5 ; i++ ) {
        h=insertList( i, h );}
    return h ;
}
int main( void ) {
    List *z = somelist() ;
}
```



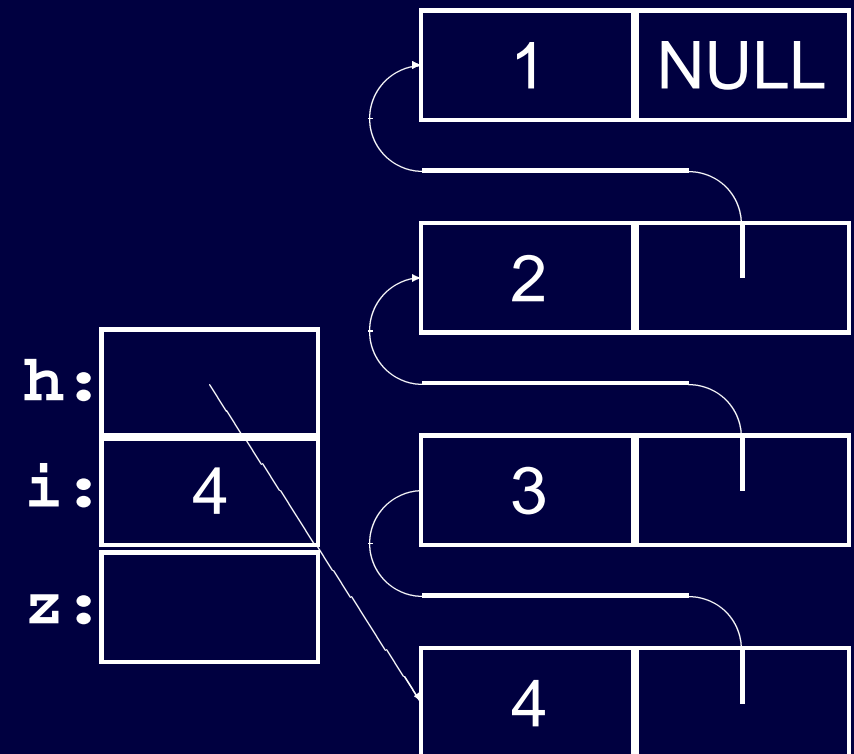
Example of a list

```
#include <stdlib.h>
typedef struct L { int x ; struct L *next ; } List;
List *insertList( int hd, List *tl ) {
    List *t = calloc( 1, sizeof( List ) ) ;
    t->x = hd ; t->next = tl ; return t ;
}
List *somelist( void ) {
    List *h = NULL ; int i ;
⇒ for( i=1 ; i<5 ; i++ ) {
    h=insertList( i, h );}
    return h ;
}
int main( void ) {
    List *z = somelist() ;
}
```



Example of a list

```
#include <stdlib.h>
typedef struct L { int x ; struct L *next ; } List;
List *insertList( int hd, List *tl ) {
    List *t = calloc( 1, sizeof( List ) ) ;
    t->x = hd ; t->next = tl ; return t ;
}
List *somelist( void ) {
    List *h = NULL ; int i ;
    for( i=1 ; i<5 ; i++ ) {
        h=insertList( i, h );}
    => return h ;
}
int main( void ) {
    List *z = somelist() ;
}
```

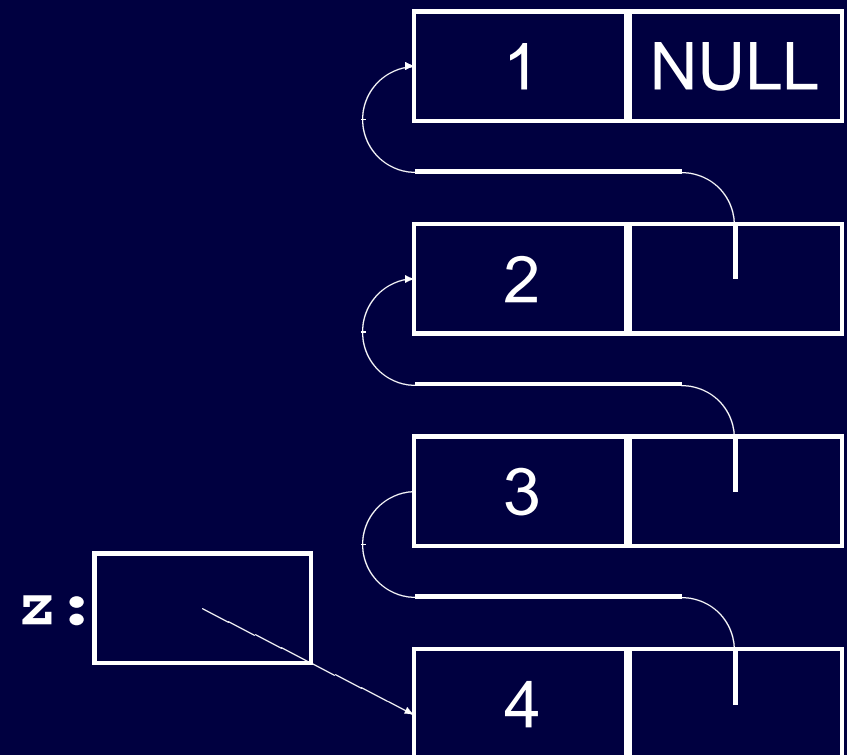


Example of a list

```
#include <stdlib.h>
typedef struct L { int x ; struct L *next ; } List;
List *insertList( int hd, List *tl ) {
    List *t = calloc( 1, sizeof( List ) ) ;
    t->x = hd ; t->next = tl ; return t ;
}
```

```
List *somelist( void ) {
    List *h = NULL ; int i ;
    for( i=1 ; i<5 ; i++ ) {
        h=insertList( i, h );}
    return h ;
}
```

```
int main( void ) {
    List *z = somelist() ;
```

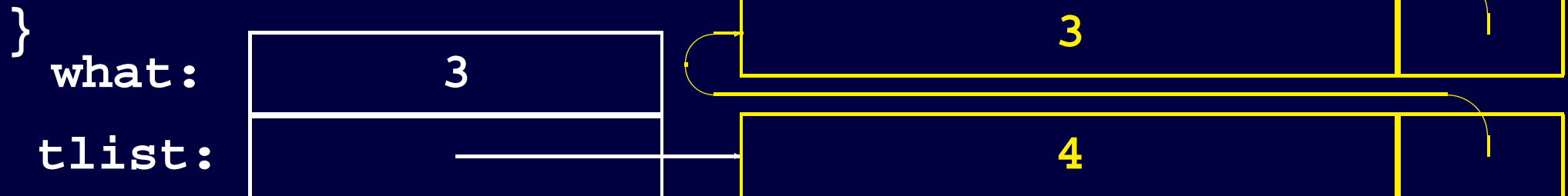


Finding an element, while

```
typedef struct L {
    int x ;
    struct L *next ;
} List ;
List *find( int what, List *tlist ) {
    while( tlist != NULL ) {
        if( tlist->x == what ) {
            return tlist ;
        }
        tlist = tlist->next ;
    }
    return NULL ;
}
```

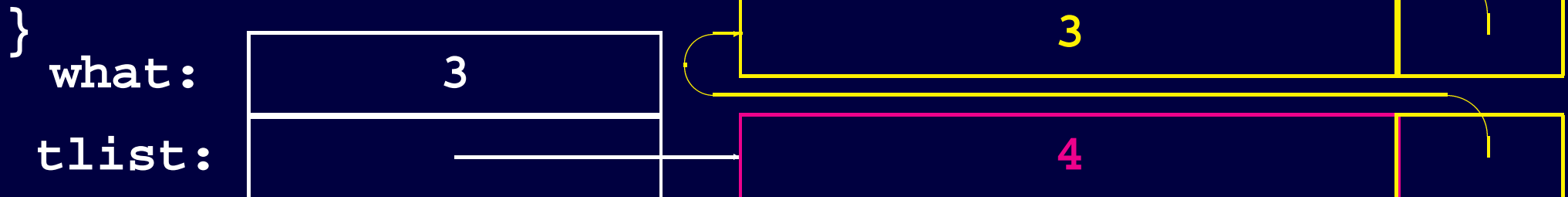
Finding an element, while

```
typedef struct L {
    int x ;
    struct L *next ;
} List ;
List *find( int what, List *tlist ) {
⇒ while( tlist != NULL ) {
    if( tlist->x == what ) {
        return tlist ;
    }
    tlist = tlist->next ;
}
return NULL ;
```



Finding an element, while

```
typedef struct L {  
    int x ;  
    struct L *next ;  
} List ;  
List *find( int what, List *tlist ) {  
    while( tlist != NULL ) {  
⇒     if( tlist->x == what ) {  
        return tlist ;  
    }  
    tlist = tlist->next ;  
}  
return NULL ;
```



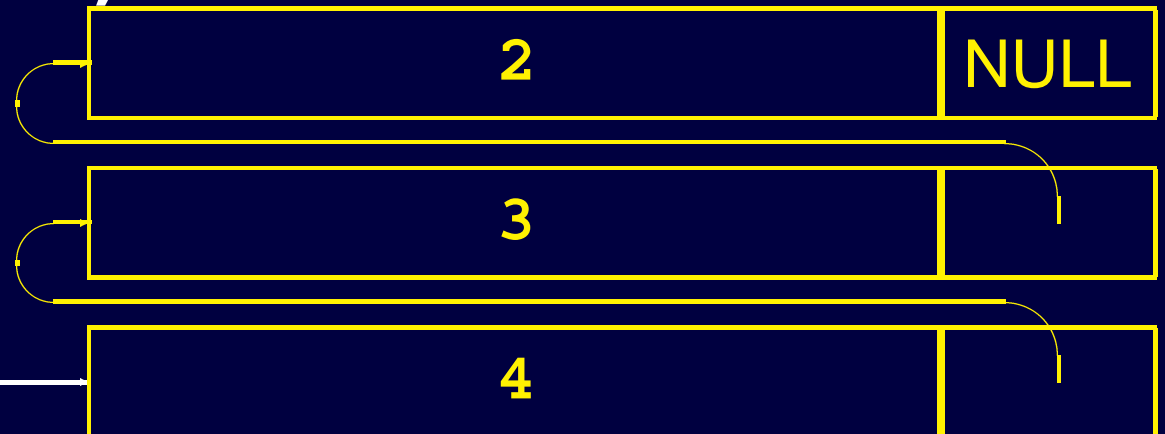
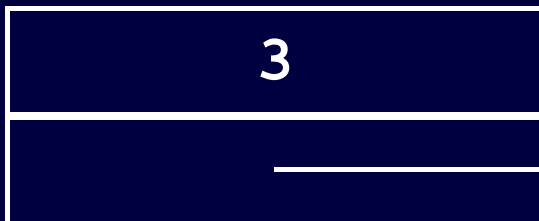
Finding an element, while

```
typedef struct L {
    int x ;
    struct L *next ;
} List ;
List *find( int what, List *tlist ) {
    while( tlist != NULL ) {
        if( tlist->x == what ) {
            return tlist ;
        }
        tlist = tlist->next ;
    }
    return NULL ;
}
```

what:

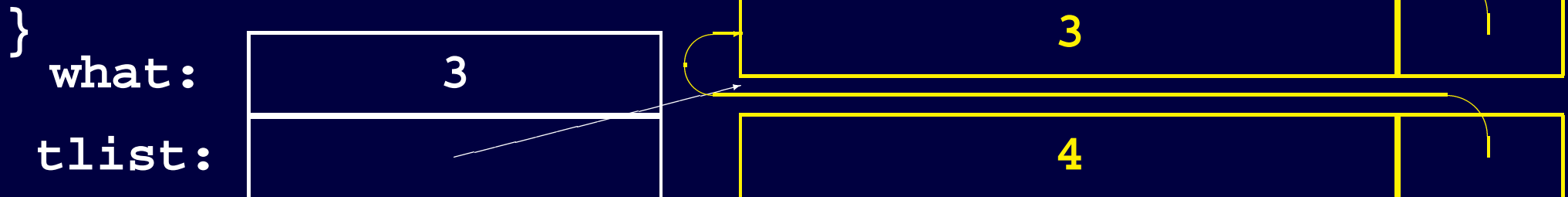
3

tlist:



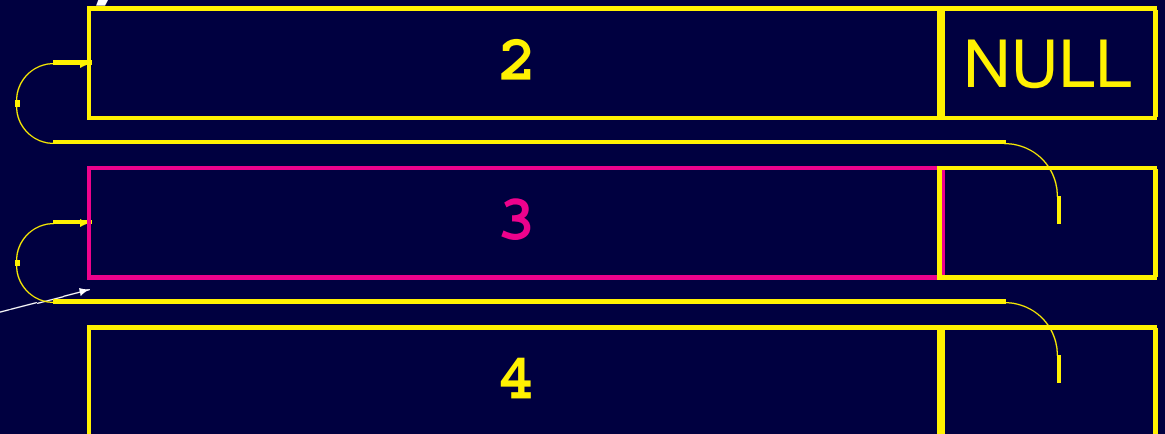
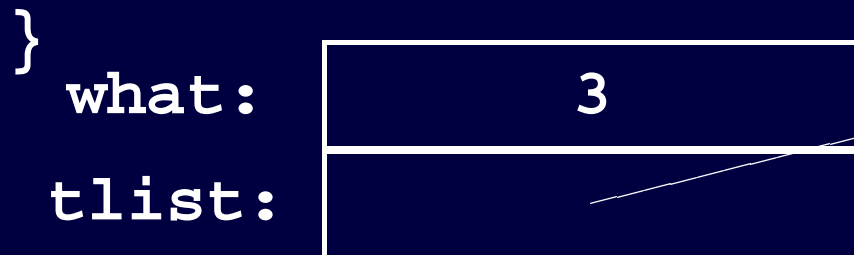
Finding an element, while

```
typedef struct L {
    int x ;
    struct L *next ;
} List ;
List *find( int what, List *tlist ) {
⇒ while( tlist != NULL ) {
    if( tlist->x == what ) {
        return tlist ;
    }
    tlist = tlist->next ;
}
return NULL ;
```



Finding an element, while

```
typedef struct L {
    int x ;
    struct L *next ;
} List ;
List *find( int what, List *tlist ) {
    while( tlist != NULL ) {
        ⇒ if( tlist->x == what ) {
            return tlist ;
        }
        tlist = tlist->next ;
    }
    return NULL ;
}
```



Finding an element, while

```
typedef struct L {
    int x ;
    struct L *next ;
} List ;
List *find( int what, List *tlist ) {
    while( tlist != NULL ) {
        if( tlist->x == what ) {
            return tlist ;
        }
        tlist = tlist->next ;
    }
    return NULL ;
}
```

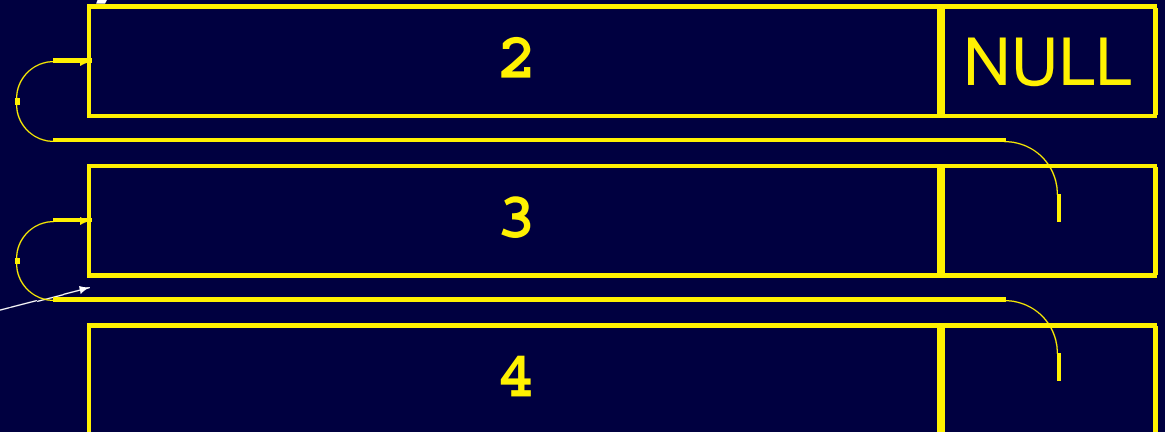
⇒

what:

3

tlist:

--



Finding an element, for-loop

```
typedef struct L {
    int x ;
    struct L *next ;
} List ;
List *find( int what, List *tlist ) {
    for( ; tlist != NULL ; tlist = tlist->next ) {
        if( tlist->x == what ) {
            return tlist ;
        }
    }
    return NULL ;
}
```

Adding to the end of a list

```
typedef struct L { int x ; struct L *next ; } List ;

List *addList( List *tlist, int element ) {
    if( tlist == NULL ) {
        return insertList( element, NULL ) ;
    } else {
        List *t ;
        for( t = tlist ; t->next != NULL ; t = t->next ) {
        }
        t->next = insertList( element, NULL ) ;
        return tlist ;
    }
}
```

Adding to the end of a list

```
typedef struct L { int x ; struct L *next ; } List ;
typedef struct {
    List *head ;
    List *tail ;
} FullList ;
```

```
FullList addFullList( FullList c, int element ) {
    if( c.head == NULL ) {
        c.head = c.tail = insertList( element, NULL ) ;
    } else {
        c.tail->next = insertList( element, NULL ) ;
        c.tail = c.tail->next ;
    }
    return c ;
}
```

Inserting in front of a list

```
typedef struct L { int x ; struct L *next ; } List ;
typedef struct {
    List *head ;
    List *tail ;
} FullList ;
```

```
FullList insertFullList( FullList c, int element ) {
    if( c.head == NULL ) {
        c.head = c.tail = insertList( element, NULL ) ;
    } else {
        c.head = insertList( element, c.head ) ;
    }
    return c ;
}
```

Reusing dynamic memory

The function `calloc` allocates memory

- When you don't need it anymore, you will have to dispose of it.
- In order to recycle it, you must call `free`.
- The system will then earmark the memory to be reused.

```
List *p = calloc( 1, sizeof( List ) ) ;  
...  
free( p ) ;
```

⇒ Do not use the memory after you called `free`

⇒ Do not call `free` twice on the same memory block.

Destructor for lists

How do you throw a whole list away?

- When you have thrown the first cell away, you don't know the rest anymore...
- Remember the next cell, before destroying the first:

```
destroyList( List *s ) {  
    List *remember ;  
    while( s != NULL ) {  
        remember = s->next ;  
        free( s ) ;  
        s = remember ;  
    }  
}
```

First remember the next pointer, then destroy

Destructor for lists

How do you throw a whole list away?

- When you have thrown the first cell away, you don't know the rest anymore...
- Remember the next cell, before destroying the first:

```
destroyList( List *s ) {  
    List *remember ;  
    while( s != NULL ) {  
        remember = s->next ;  
        free( s ) ;  
        s = remember ;  
    }  
}
```

First remember the next pointer, then destroy

Destructor for lists

How do you throw a whole list away?

- When you have thrown the first cell away, you don't know the rest anymore...
- Remember the next cell, before destroying the first:

```
destroyList( List *s ) {  
    List *remember ;  
    while( s != NULL ) {  
  
        free( s ) ;  
        s = s->next ;  
    }  
}
```

This will *not* work

Common functions on lists

Constructor

- Make a list node, insert it in front.

Operations

- Head (look at the first element)
- Tail (returns everything but the last element)
- Many more, will be defined later on

Destructor

- Free the memory.

This division will be seen more often! Every data structure has constructors, destructors, and operations

⇒ Abstract Data Type, or ADT

Arrays versus Lists

Similarities between lists and arrays

- Both store a sequence of values
- Both sequences of identical types

What are the differences?

- Efficiency
- Finding element 5 in an array is an immediate operation
- Finding element 5 in a list takes me 5 steps

Arrays versus Lists

Similarities between lists and arrays

- Both store a sequence of values
- Both sequences of identical types

What are the differences?

- Efficiency
- Finding element 9999 in an array is an immediate operation
- Finding element 9999 in a list takes me 9999 steps

Arrays versus Lists

Similarities between lists and arrays

- Both store a sequence of values
- Both sequences of identical types

What are the differences?

- Efficiency
- Finding element 9999 in an array is an immediate operation
- Finding element 9999 in a list takes me 9999 steps
- Inserting an element in front of a list is cheap
- Inserting an element in an array is expensive

Arrays versus Lists II

For a list/array of length n , the costs are:

Operation	List	Array
Create n elements	n steps	n steps
Access first element	1 step	1 step
Access element i	i steps	1 step
Insert new first element	1 steps	n steps
Insert new element at i	i step	n steps
Delete first element	1 step	n steps
Delete element at i	i steps	n steps

Coming lectures

- Grammar
- Ethics
- Debugging