

# Tipus no elementals

- Els tipus no elementals representen dades.
- Quan les dades d'un conjunt són totes del mateix tipus, es parla de taules (vectors i matrius).
- Si les dades poden ser de diferents tipus, s'utilitzen les estructures de C (**struct**).

# Definició de nous tipus de dades

- Un nou tipus en C es defineix com:

```
typedef tipus nom_nou_tipus;
```

- `typedef int T[20];`

Defineix el tipus `T` com un vector de 20 `int`

- Si es vol definir el tipus `Pt2D` com dos valors `double` i un `int` “agrupats”

```
typedef struct {  
    double x, y;  
    int c;  
} Pt2D;
```

# Struct: definició

- Conjunt de dades que no tenen perquè ser totes del mateix tipus.
- Cada element s'anomena camp.
- En C, per definir un **struct** es pot fer de dues maneres:
  - definint un nou tipus amb la instrucció **typedef**
  - definint directament l'estructura

## Struct: definició mitjançant un nou tipus

- Declaració d'un nou tipus **struct** (nom intern opcional):

```
typedef struct [nom_intern] {
    tipus1 camp1;
    tipus2 camp2;
    ...
    tipusN campN;
} nom_nou_tipus;
```

- Després del **typedef** podem declarar variables o apuntadors de la manera usual

```
nom_nou_tipus nom_variable;
nom_nou_tipus *nom_punter;
```

- Per accedir als camps d'un **struct**:

```
nom_variable.nom_camp
*nom_punter.nom_camp
nom_punter->nom_camp
```

# Struct: definició

- Definició d'un punt i color:

```
typedef struct {  
    float x,y;  
    int R,G,B;  
    int selecc;  
} punt_i_Color;
```

- Declaració de variables del nou tipus

```
punt_i_Color p1, p2;  
punt_i_color *punter;
```

- Accés a un camp o a tota l'estructura

```
p1.selecc = 1;  
p2.x = 0.0;  
punter = &p1;  
punter->y = punter->x;
```

# Exemple

```
typedef struct Node_t {  
    int id;          /* ident. del node */  
    char nom[30];    /* info del node */  
} Node;
```

# Struct: definició directa

## ■ Definició

```
struct nom_struct {  
    tipus1 camp1;  
    tipus2 camp2;  
    ...  
    tipusN campN;  
};
```

- Declaració d'una variable de tipus **struct**. En aquest cas sempre s'ha de posar la paraula reservada **struct** davant del tipus.

```
struct nom_struct nom_variable;  
struct nom_struct *nom_punter;
```

- Per accedir als camps d'un **struct**:

```
nom_variable.nom_camp  
*nom_punter.nom_camp  
nom_punter->nom_camp
```

## Struct: definició directa

- Definició d'un punt i color:

```
struct Pt2D{  
    float x,y;  
    int R,G,B;  
    int selecc;  
};
```

- Declaració d'una variable de tipus **struct**

```
struct Pt2D p1, p2;  
struct Pt2D *p;
```

- Accés a un camp o a tota l'estructura:

```
p1.selecc = 1;  
p2.x = 0.0;  
p = &p2;  
p->x = p->x + 2;
```



# Struct: Normes d'ús

- Es poden assignar directament tot el contingut d'un **struct** en un altre **struct** del mateix tipus.

```
p1 = p2;  
p = &p1;
```

- Les regles per a passar paràmetres de tipus **struct** a funcions són les mateixes que en **tipus elementals**: si es vol modificar el contingut de l'estructura dins de la funció, es passa el punter a l'estructura, en cas contrari, es passa directament l'estructura.

# Exemple

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

#define N 20

typedef long int tauFib[N];

typedef struct {
    tauFib t;
    int n;
} taula;

void escriureFib( taula );
```

## Exemple: cont

```
int main (void) {  
    taula fib;  
    int i,j;  
  
    fib.t[0] = 0;  
    fib.t[1] = 1;  
    fib.n = 2;  
    while (fib.n < N ) {  
        fib.t[fib.n] = fib.t[fib.n-1] + fib.t[fib.n-2];  
        ++(fib.n);  
    }  
    printf("Nombres_de_Fibonacci\n");  
    escriureFib(fib);  
}
```

## Exemple: cont

```
for (i = 0; i < fib.n; ++i) {  
    if( fib.t[i] % 7 == 0) {  
        for (j = i + 1; j < fib.n; ++j) {  
            fib.t[j-1] = fib.t[j];  
        }  
        --(fib.n);  
    }  
}  
printf("Nombres_de_Fibonacci,");  
printf("_sense_multiples_de_7\n");  
escriureFib(fib);  
return 0;  
}
```

## Exemple: cont

```
void escriureFib(taula f) {  
    int i;  
    for ( i = 0; i < f.n; ++i ) {  
        printf("_%ld", f.t[i]);  
    }  
    printf("\n");  
    return;  
}
```

# Exemple més sofisticat

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

typedef struct {
    long int *t;
    int n;
} successio;

void escriureSucc( successio );
successio * creaSucc (int);
successio maxSucc(successio, successio);
```

## Exemple

```
int main (void) {
    successio *s1, *s2, s3;
    int nTer;
    srand( (unsigned) time(NULL) );
    printf("N.termes_max?");
    scanf("%d", &nTer);
    s1 = creaSucc(4+rand()%nTer);
    s2 = creaSucc(4+rand()%nTer);
    s3 = maxSucc(*s1,*s2);
    printf("Successio_1\n"); escriureSucc(*s1);
    printf("Successio_2\n"); escriureSucc(*s2);
    printf("Successio_3\n"); escriureSucc(s3);
    free (s3.t); free (s2->t); free (s1->t);
    free (s2); free (s1);
    return 0;
}
```

## Exemple

```
successio * creaSucc (int n) {
    int i;
    successio *s;

    s = (successio *) malloc (sizeof(successio));
    if (s == NULL) return s;

    s->t = (long int *) malloc (n*sizeof(long int));
    if (s->t == NULL) { free (s); return NULL; }
    s->n = n;

    s->t[0] = rand()%10; s->t[1] = rand()%10;
    for (i = 2; i < s->n; ++i) {
        s->t[i] = - s->t[i-1] + (rand()%10)*s->t[i-2];
    }
    return s;
}
```



## Exemple

```

successio maxSucc(successio s1, successio s2) {
    int j, n;
    successio s, *psL, *psC;
    n = ( s1.n > s2.n ? s1.n : s2.n);
    if ( n == s1.n ) { psL = &s1; psC = &s2;
    } else { psL = &s2; psC = &s1;
    }
    s.t = (long int *) malloc (n*sizeof(long int));
    if (s.t == NULL) { s.n = 0; return s; }
    s.n = n;
    for (j = 0; j < psC->n ; ++j)
        s.t[j]=(s1.t[j]>s2.t[j]?s1.t[j]:s2.t[j]);
    for (j = psC->n; j < psL->n ; ++j)
        s.t[j] = psL->t[j];
    return s;
}

```

# Exemple

```
void escriureSucc(successio f) {  
    int i;  
    if (f.n == 0) {  
        printf("Succ._buida!\n");  
        return;  
    }  
    for ( i=0; i< f.n; ++i) {  
        printf("_%ld",f.t[i]);  
    }  
    printf("\n");  
    return;  
}
```

# Sortida

```
N.termes max?10
```

```
Successio 1
```

```
4 8 -8 16 -56 184 -688 1792 -5232 14192 -14192 113536
```

```
Successio 2
```

```
1 5 -4 39 -59 254 -667 2953 -6288 21053
```

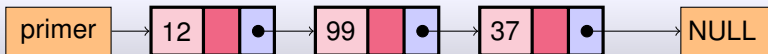
```
Successio 3
```

```
4 8 -4 39 -56 254 -667 2953 -5232 21053 -14192 113536
```

# Llistes

- Els vectors ofereixen accés ràpid als elements degut a que són col·locats en memòria contiguament.
- Les **l·listes encadenades** són entitats de programació que guarden la informació en nodes, i cada node conté, a banda de la seva pròpia informació un apuntador al node següent.
- Al primer node l'anomenarem **primer** o **cap** i al darrer **últim** o **cua**.

En C s'implementen usant estructures i apuntadors.



# Llistes

## Avantatges:

- La memòria es pot reservar i alliberar segons necessitat. A vegades els vectors estan sobredimensionats amb molts elements no usats.
- Alguns algoritmes són més ràpids que en versió “vector”, en especial si cal canviar de mida sovint.
- Permeten implementar estructures de dades no lineals usant nodes amb més apuntadors.  
Exemples: arbres o grafs.

## Desavantatges:

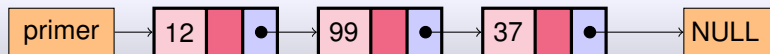
- Cada node té un espai suplementari pels apuntadors (4 o 8 bytes per cada apuntador)
- Alguns algoritmes són més lents que en versió “vector”.

# Llistes

```
typedef struct Node_t *pNode;

typedef struct Node_t {
    int id;           /* ident. del node, unica */
    char nom[30];     /* info. del node, el que calgui */
    pNode seguent;    /* punter a un altre node */
} Node;

pNode primer;
```



# Llistes: Operacions

La creació del primer node és senzilla:

```
primer = (Node *) malloc(sizeof(Node));  
  
/* Omplir id (12) i info */
```



# Llistes: Operacions

La creació del primer node és senzilla:

```
primer = (Node *) malloc(sizeof(Node));
```

```
/* Omplir id (12) i info */
```

```
primer -> seguent = NULL;
```



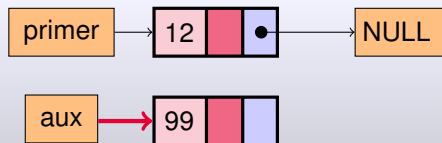


# Llistes: Operacions

Afegir un segon node també ho és:

```
aux = (Node *) malloc(sizeof(Node));
```

```
/* Omplir id (99) i info */
```



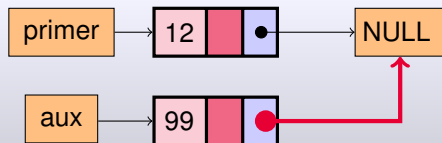
# Llistes: Operacions

Afegir un segon node també ho és:

```
aux = (Node *) malloc(sizeof(Node));
```

```
/* Omplir id (99) i info */
```

```
aux -> seguent = NULL;
```



# Llistes: Operacions

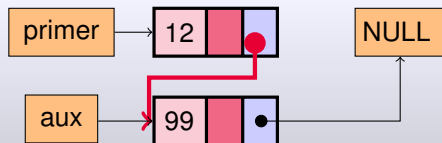
Afegir un segon node també ho és:

```
aux = (Node *) malloc(sizeof(Node));
```

```
/* Omplir id (99) i info */
```

```
aux -> seguent = NULL;
```

```
primer -> seguent = aux;
```

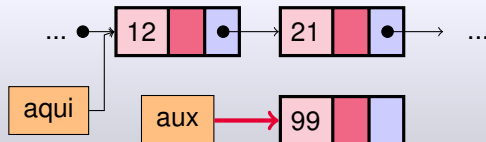


# Llistes: Operacions

Afegir **darrera** un node: cal buscar i tenir-lo apuntat (aqui)

```
aux = (Node *) malloc(sizeof(Node));
```

```
/* Omplir id (99) i info */
```



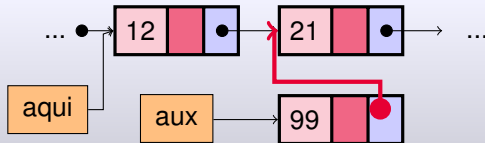
# Llistes: Operacions

Afegir **darrera** un node: cal buscar i tenir-lo apuntat (**aqui**)

```
aux = (Node *) malloc(sizeof(Node));
```

```
/* Omplir id (99) i info */
```

```
aux -> seguent = aqui -> seguent;
```



# Llistes: Operacions

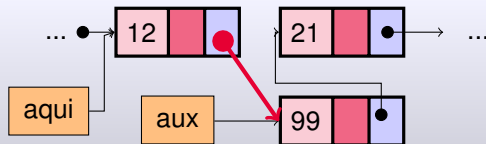
Afegir **darrera** un node: cal buscar i tenir-lo apuntat (aqui)

```
aux = (Node *) malloc(sizeof(Node));
```

```
/* Omplir id (99) i info */
```

```
aux -> seguent = aqui -> seguent;
```

```
aqui -> seguent = aux;
```

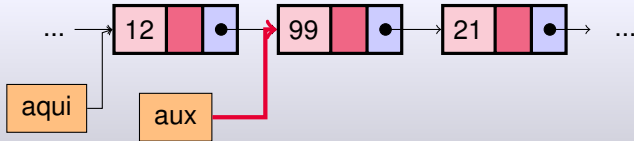


# Llistes: Operacions

Extreure **darrera** d'un node: cal buscar i tenir-lo apuntat (**aqui**)

```
/* Extreure id (99) */
```

```
aux = aqui -> seguent;
```

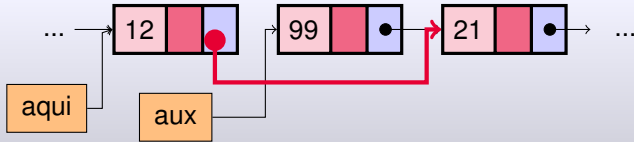


# Llistes: Operacions

Extreure **darrera** d'un node: cal buscar i tenir-lo apuntat (**aqui**)

```
/* Extreure id (99) */
```

```
aux = aqui -> seguent;  
aqui -> seguent = aux -> seguent;
```





# Llistes: Operacions

Extreure **darrera** d'un node: cal buscar i tenir-lo apuntat (**aquí**)

```
/* Extreure id (99) */
```

```
aux = aqui -> seguent;  
aqui -> seguent = aux -> seguent;  
aux -> seguent = NULL;
```

