- 8° Modulo parte 1
  - TAD Tipi Astratti di Dati
    - Concetti generali
    - Es. matrici sparse, tabelle
  - ______________________
  - ______________________
  - ______________________

http://www.elet.polimi.it/upload/martucci/index.html

Strutture di dati e loro realizzazione in C

Tipo di dato astratto:

<S, Op, C>

- Un insieme di insiemi S
- Un insieme di operazioni che si applicano a valori del dominio o che hanno come risultato valori del dominio (Op);
- Un insieme di costanti che denotano valori del dominio (C).

Tipo concreto:

Termine usato per indicare un tipo di dato presente in un linguaggio di programmazione.

Strutture dati:

Termine usato per indicare tipi con elementi del dominio compositi (ad esempio, vettori, liste, tavole).

Ad esempio:

- vettori e **matrici**;
- tavole;
- **liste**;
- insiemi;
- pile e code;
- **alberi** e grafi.

Struttura dati

-

- la dimensione della struttura dati non è nota a priori e varia dinamicamente durante l'esecuzione del programma
- se vengono usate variabili statiche (cioè **dichiarate** globalmente o localmente) è necessario prevedere una dimensione massima. Il compilatore, prima dell'esecuzione, determina la quantità di memoria da allocare, che in esecuzione può risultare molto sottoutilizzata.

Nei linguaggi di programmazione d'alto livello è, in generale, disponibile un meccanismo che consente di **riservare memoria per una variabile durante l'esecuzione**, cioè di **crearla dinamicamente**. E' ovviamente disponibile anche il meccanismo che consente di rilasciare (rendere nuovamente disponibile) la memoria allocata.

Matrici: ottimizzazione dell'occupazione di memoria

La rappresentazione di matrici e vettori richiede una occupazione di memoria ***proporzionale*** alla ***dimensione*** (numero dei componenti) della struttura dati.

Per ridurre lo spazio allocato:

Rappresentazione compatta di matrici

- e` utilizzata se la maggior parte dei valori memorizzati nella matrice e` uguale ad un valore predominante (***matrice sparsa***).

- "Costruibile" in qualunque linguaggio di programmazione di alto livello (ad esempio, C, Pascal, FORTRAN).

Rappresentazione compatta di matrici

Matrice N×M occupa N×M locazioni di memoria.

Matrice Sparsa:

Se molti elementi hanno lo stesso valore (**valore predominante**) si potrebbero rappresentare in modo piu` **compatto** risparmiando memoria.

Primo metodo:

- Si memorizza la terna:

  <dim-riga,dim-colonna,valore-predominante>

- Si rappresentano tutti i valori significativi mediante la terna:

  <indice-riga,indice-colonna,valore>

- Realizzazione mediante una matrice a tre colonne.

Esempio:

Data la matrice sparsa:

8	**0**	**0**	**11**	**0**	**0**
0	**0**	**21**	**0**	**0**	**0**
0	**0**	**0**	**15**	**0**	**0**
0	**0**	**3**	**0**	**0**	**0**

Memorizzazione compatta:

Riga	Colonna	Valore	
4	**6**	**0**	(dim. e val. predominante)
1	**1**	**8**	
1	**4**	**11**	
2	**3**	**21**	
3	**4**	**15**	
4	**3**	**3**	
10	**10**	**0**	

Dimensioni: 3×(n-valori)

Risparmio di occupazione di memoria:

- 21 locazioni invece di 24 ... la differenza cresce con

Matrici Sparse

Correttezza della rappresentazione:

Dato un valore della matrice sparsa, esiste un valore della matrice compatta che lo rappresenta se e solo se il numero dei componenti della matrice compatta e` sufficiente alla memorizzazione degli elementi diversi dal valore predominante.

Gestione:

lettura: ***accedi(M, i, j)***
scrittura: ***memorizza(M, i, j,val)***

➣ si ha un risparmio di memoria, ma **un costo piu` alto di esecuzione** per le istruzioni di lettura e scrittura.

Ad esempio:

- nel caso peggiore la funzione ***accedi*** deve accedere a tutti gli elementi della matrice compatta.
- La scansione degli elementi per colonna e` anch'essa costosa.

Esistono altri metodi che facilitano la scansione per riga o per colonna.

Matrici Sparse

Rappresentazione con vettore di accesso:

Facilita la scansione per riga: si associa alla rappresentazione compatta della matrice un vettore di accesso R:

- Il numero di componenti del vettore di accesso R e' pari al numero di righe della matrice sparsa.

- La componente I-esima del vettore di accesso indica la prima posizione della matrice compatta in cui si trova un elemento della matrice sparsa il cui ***indice di riga*** e` I.

- Per accedere all’elemento della matrice sparsa di indici <i,j> si accedera` a R(i): il valore di R(i) indica da dove iniziare la scansione della matrice compatta.

Esempio:

8	**0**	**0**	**11**	**0**	**0**
0	**0**	**21**	**0**	**0**	**0**
0	**0**	**0**	**15**	**0**	**0**
0	**0**	**3**	**0**	**0**	**0**

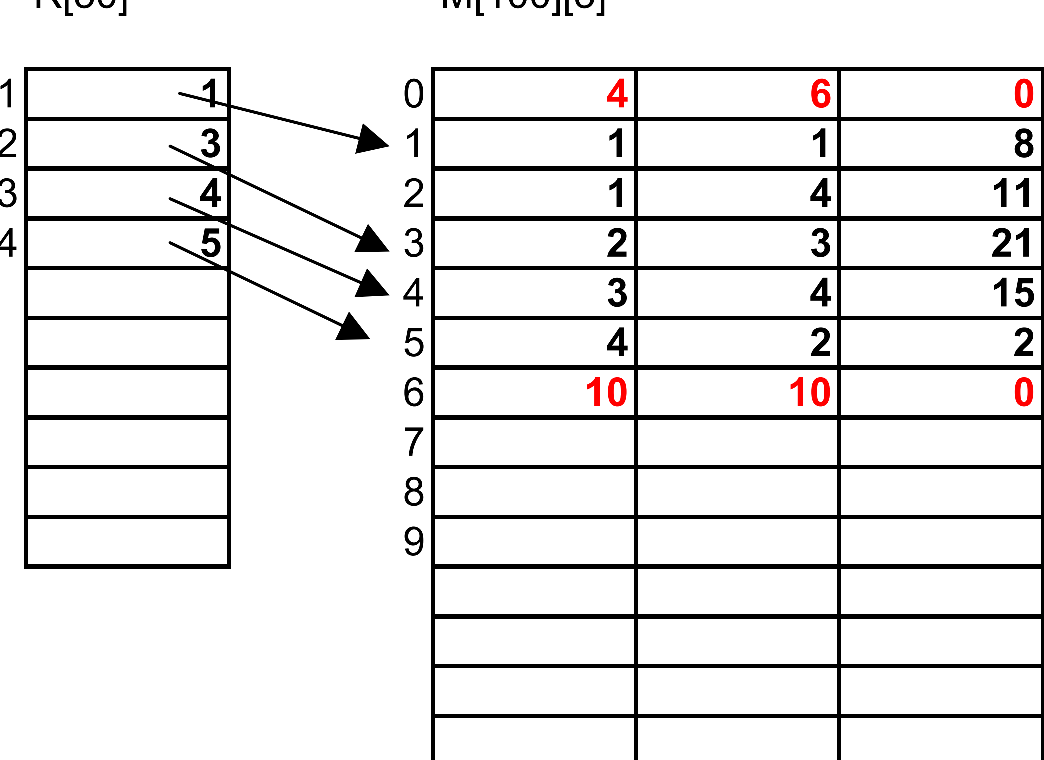
Memorizzazione:

Riga	Colonna	Valore
4	**6**	**0**
1	**1**	**8**
1	**4**	**11**
2	**3**	**21**
3	**4**	**15**
4	**3**	**3**
10	**10**	**0**

R
1
3
4
5

Esercizio:

Realizzare questa modalita` di rappresentazione per matrici sparse con le relative procedure.



R[30]
M[100][3]
1 1
2 3
3 4
4 5
0 4 6 0
1 1 1 8
2 1 4 11
3 2 3 21
4 3 4 15
5 4 2 2
6 10 10 0
7
8
9

Tavole

Una tavola e` un tipo di dato astratto per rappresentare insiemi di coppie ***<chiave, attributi>***.

Ciascuna coppia rappresenta dati riferiti ad un' unica entita` logica (ad es., persona, documento, etc.) identificata in modo univoco dalla ***chiave***.

Esempio:

Rappresentazione concreta ***sequenziale***

NOME	COGNOME	REDDITO	ALIQUOTA
	...		

```
typedef struct      {char     Nome[20];
                     charCognome[20];
                     int      Reddito;
                     int      Aliquota;} Persone;

Persone[99]          Tavola;
```

Tavole

Spesso sono memorizzate su dispositivi di memoria di massa (***file***).

- In **C** si utilizzano ***file binari*** (lettura e scrittura di strutture).

Operazioni tipiche sulle tavole:

- **inserimento** di un elemento <Chiave, Attributi>

  inserisci: tavola × chiave × attributi → tavola

- **cancellazione** di un elemento (nota la chiave)

  cancella: tavola × chiave → tavola

- verifica di **appartenenza** di un elemento

  esiste: tavola × chiave → boolean

- **ricerca** di un elemento nella tavola

  ricerca: tavola × chiave → attributi

L'operazione di ***ricerca*** e\` la piu\` importante.

➣ Spesso la rappresentazione concreta viene scelta in modo da ottimizzare questa operazione.

Tipi Astratti di Dati: operazioni

- Un TAD deve essere usato SOLO attraverso le operazioni definite
- Es.
  - Inserimento
  - Cancellazione
  - Appartenenza
  - Ricerca
  - Nullità
- La complessità e la fisicità del dato sono nascoste

Strutture dati dinamiche

Spesso e` necessario gestire e manipolare in un programma collezioni di dati le cui dimensioni sono soggette a variare dinamicamente.

Ad esempio:

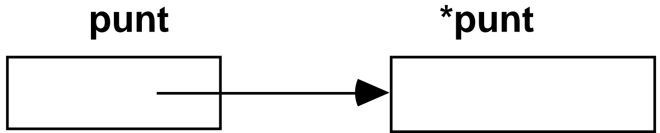
una tavola realizzata come vettore (di N componenti) non e' sufficiente se durante l'esecuzione di un programma ci si trovi a dover inserire l'(N+1)-esimo elemento.

Strutture dati dinamiche:

Il C (come altri linguaggi) consente di creare strutture dati dinamiche ***collegate*** attraverso puntatori in cui la memoria per ciascun elemento viene richiesta (ed allocata) dinamicamente (tramite la funzione ***malloc***) prelevandola dall'***area heap***.

Allocazione:

```
punt = (tipoelem *) malloc ( sizeof (tipoelem));
```



Deallocazione:

```
free (punt);
```

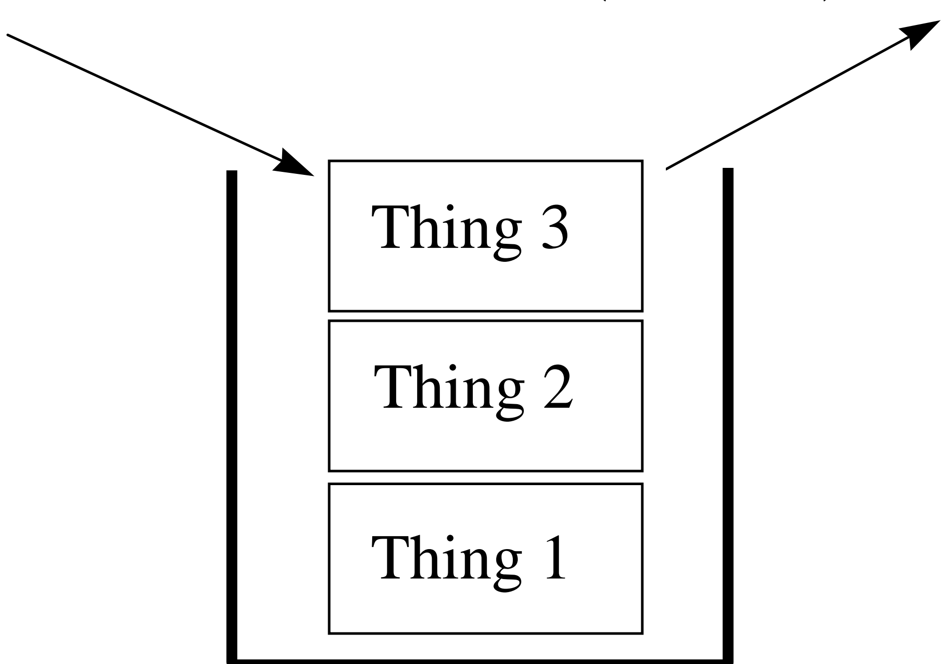
punt

- Dopo questa operazione *punt non esiste piu` (la memoria allocata per *p viene di nuovo inserita nell'insieme delle celle libere dello heap).

(memo) Stacks

A **stack** is a *logical construct* that:

- Is a “pile” or “collection” or “set” of items such that...
- Items can *only* be added to the *top*
- Items can *only* be taken off the *top*
- “Last-in-first-out” (*”LIFO”*)



- Example: dish stack in cafeteria

Tracciare l'Esecuzione con l'Activation Stack

Main this_var that_var other_var

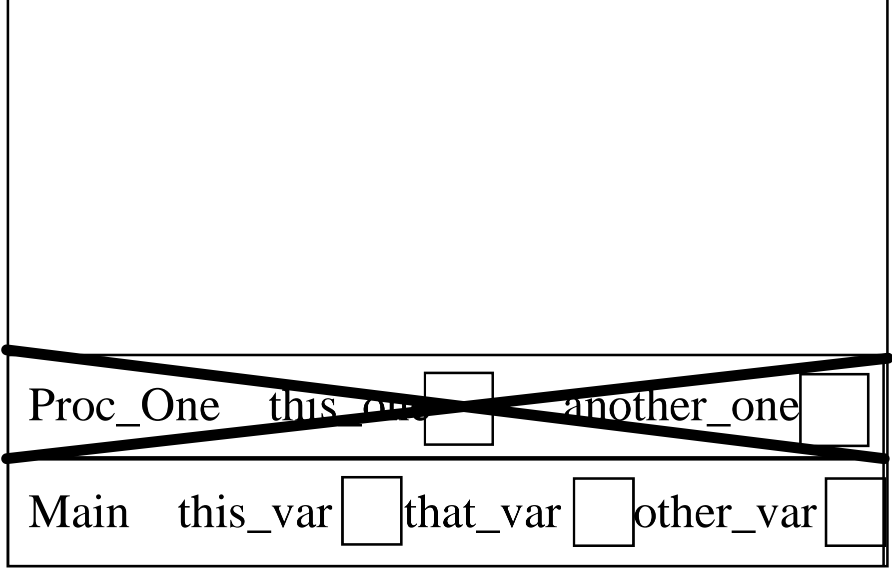
- Box represents computer memory
- Main program gets frame at bottom
- Frame sized to hold Main's variables
- More variables, bigger the frame
- In stack frame is where named variables really live

Tracing Execution with the Activation Stack

Proc_One	this_one		another_one	
Main	this_var	that_var	other_var	

- When Main calls a procedure, the procedure gets its own frame "pushed" on the stack
- That frame is where the proc's variables live
- Main "goes to sleep" until Proc_One finishes
- ONLY the top frame is active; all frames underneath it snooze

Tracing Execution with the Activation Stack



- When the procedure completes its instructions, it's frame is "popped" off the stack and *all of its data dies*
- For value of `this_one` or `another_one` to survive, they must be passed to some var that will live, i.e., one below it on the stack

How Parameters Work:
input parameters

Proc_One	this_one	4				
Main	this_var	4	that_var	7	other_var	9

if procedure's formal parameter list is:

```
void Proc_One  (int this_one) ...
```

and Main calls:

```
Proc_One(this_var);
```

- then `this_one` gets its *own copy* of the original value of `this_var`
- `this_one` cannot change value of `this_var`

How Parameters Work: *input/output* parameters

Proc_Two ret_val this_one

Main this_var 4 that_var 7 other_var 9

if the procedure's formal parameter list is:

```
int Proc_Two  (int  this_one)....
```

and Main calls:

```
this_var =  Proc_Two(that_var)
```

- then `this_one` *doesn't* gets its own copy, it gets *access to the original* instead
- changes to `this_one` really change `this_var`

Dynamic data

Example:

- We must maintain a list of data
- At some moments, the list is small
  ... so we want to use only a little memory

- At other moments, the list is large
  ... so we need to use more memory

- We need a way to *allocate* and *deallocate* data *dynamically* (i.e., *on the fly)*

- Declaring variables in the std way is *static,* won't work here: we *don't know how many* variables to declare

Introducing The Heap

12 → 18 → 21 → 23

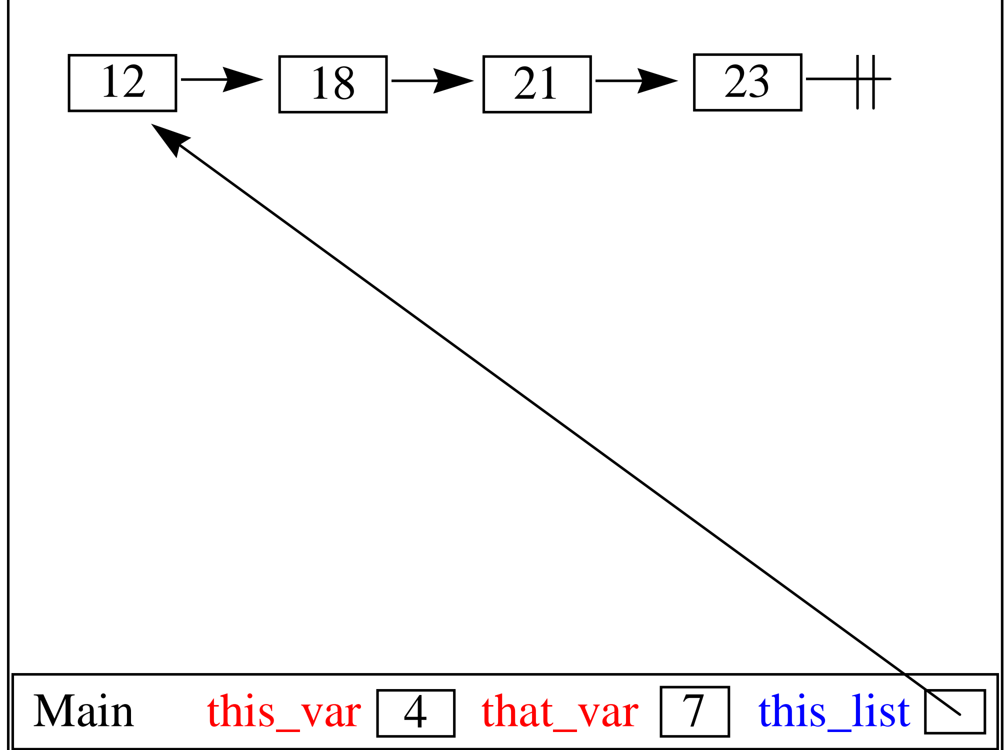
The *heap*

The *stack*

Main this_var 4 that_var 7 this_list

- The *heap* is memory not used by the *stack*
- As *stack* grows, *heap* shrinks
- *Static* variables live in the *stack*
- *Dynamic* variables live in the *heap*

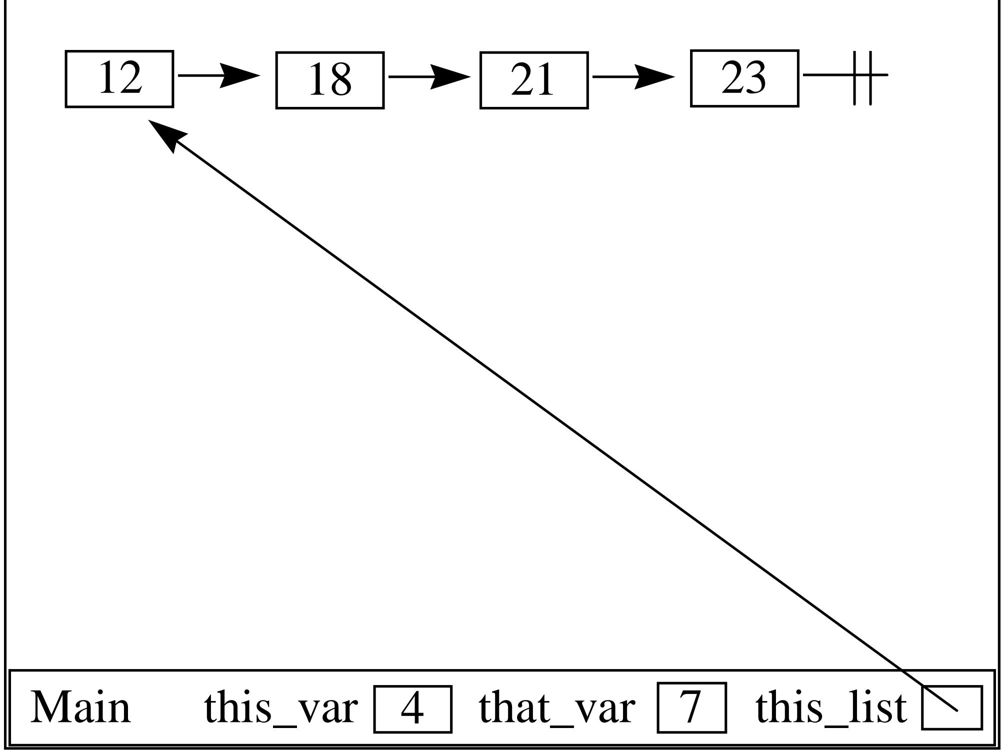
Dynamic data



- We have two static *Num* variables, plus a dynamic list... known as a *linked list*
- The list has (at the moment) 4 *nodes*
- It also has one static *pointer* variable named `this_list`

- `this_list` points to the first node
- Each node points to the next node
- The last node's pointer is *NULL*

Dynamic data



Detail view of *struct node* { ...

int data;
struct node

*next;...

Detail view of *first node's contents*:

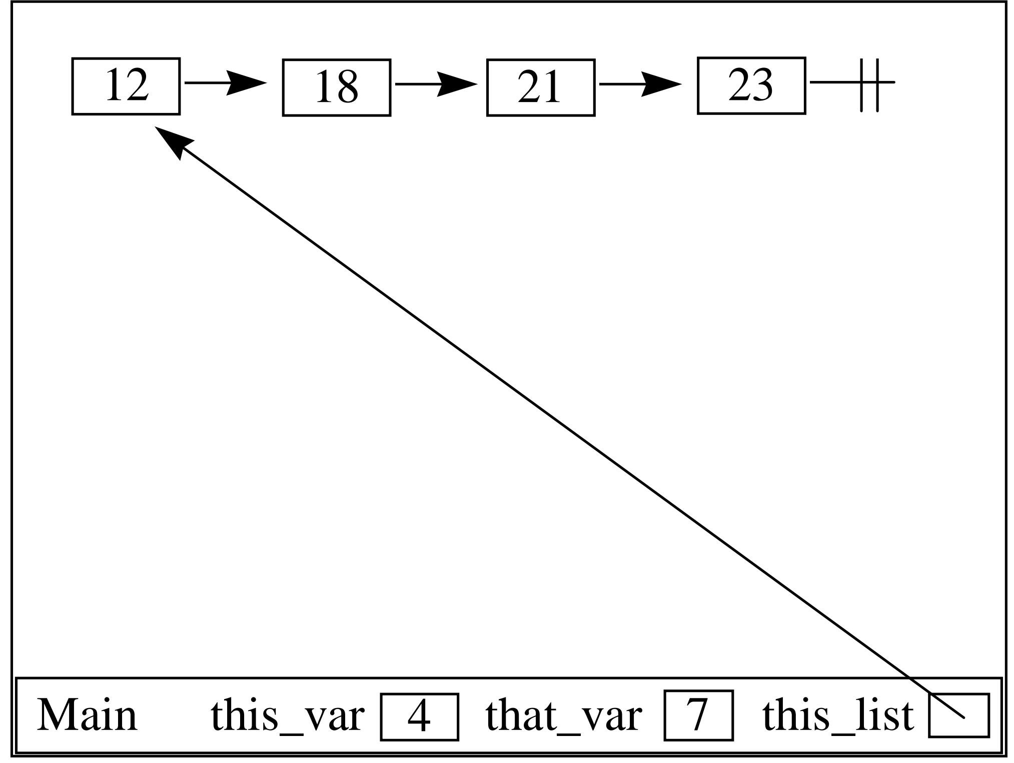
12
address of 2nd node

or

12

18

Accessing Dynamic Records via Pointers



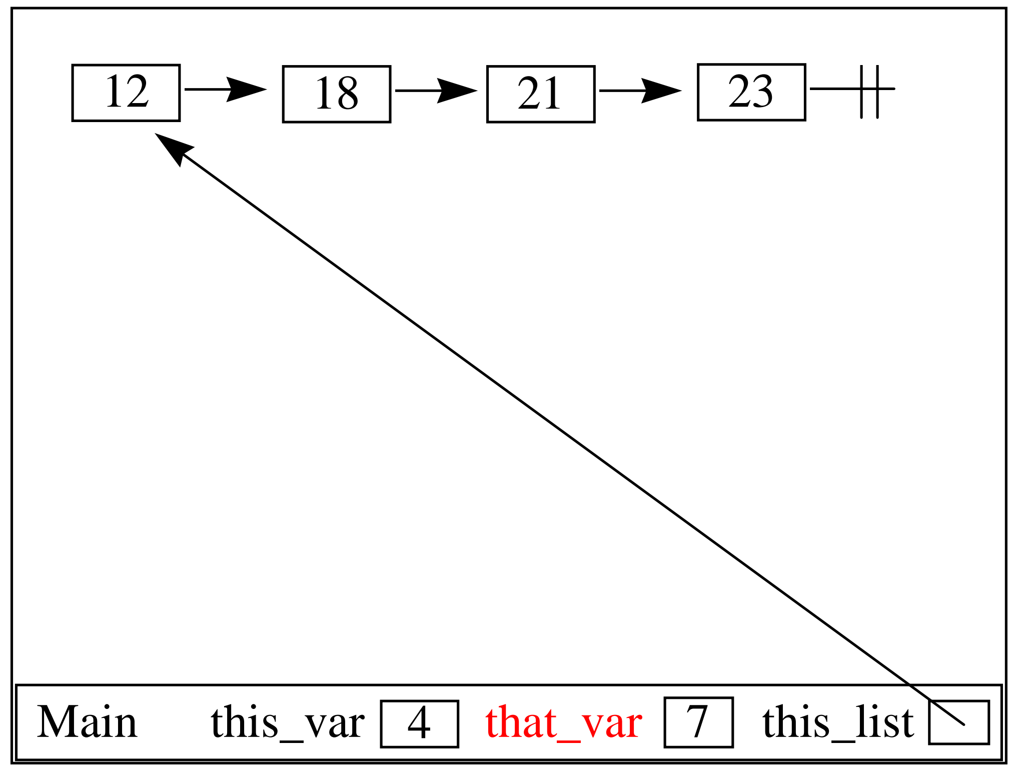
When we "follow a pointer", we say that we ***dereference*** that pointer

The (->) symbol means "dereference the pointer". . . thus . . .

```
this_list->data
```

means *"follow `this_list` to whereever it goes; once there, access the data field"*

Changing Node data Values via Pointers



Therefore:

```
that_var = this_list->data;
```

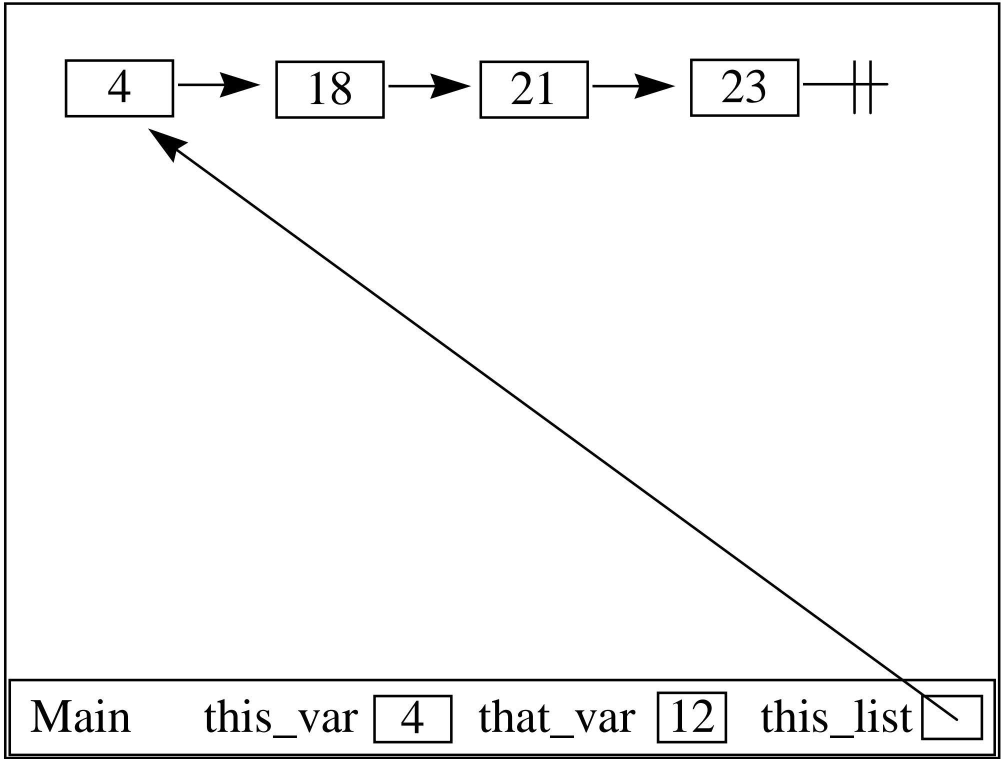
changes `that_var`'s contents from *7* to *12*

```
this_list->data = this_var;
```

changes the first node's contents from *12* to *4*

. . . giving us . . .

Changing Pointer Values

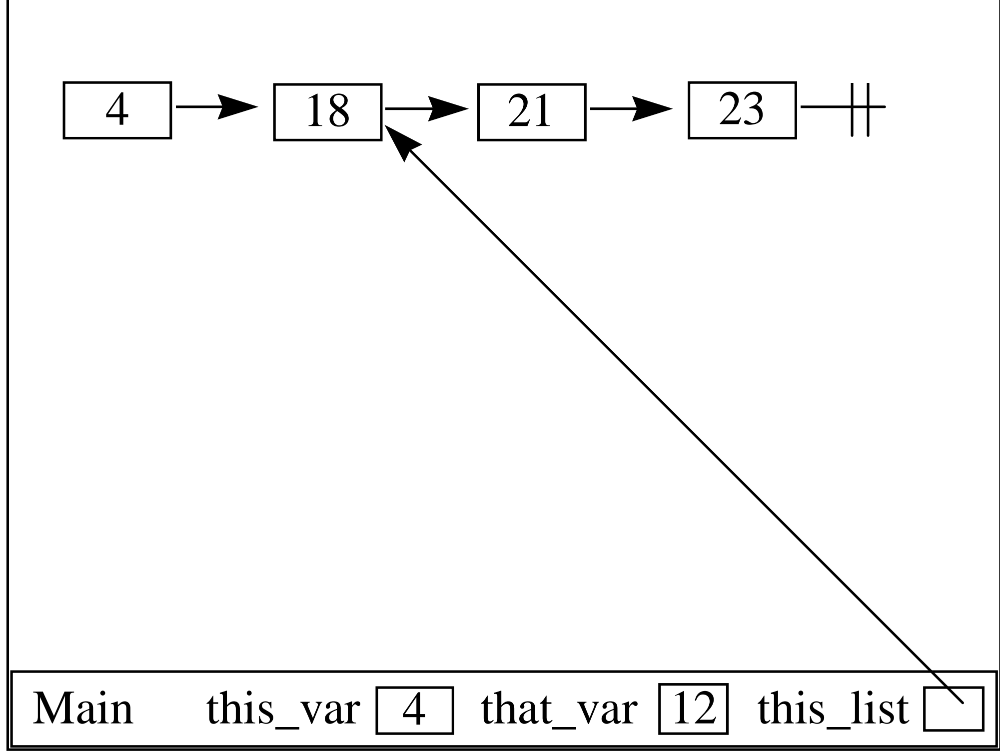


And:

```
this_list = this_list->next
```

- changes the contents of `this_list` so that it no longer tells us where the 1st node is
- instead, `this_list` now tells us where the 2nd node is . . . giving us . . .

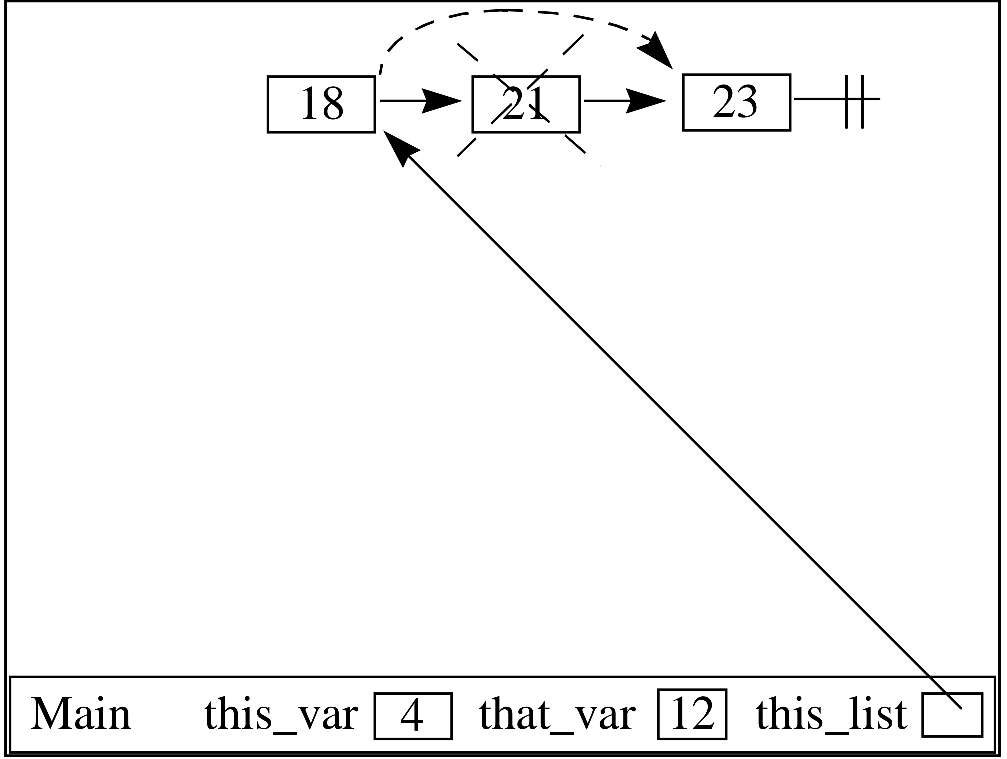
Changing Pointer Values



Note that *nothing* points to the first node

- we can no longer “get at it”: it has no pointer pointing to it
- the first node is now “lost in space”

Deallocating a Node



- With the old 1st node *killed*, the list has shrunk to 3 nodes,
- If we want to kill the node containing *21*, we arrange to have nothing pointing to it, e.g.,

```
this_list->next = this_list->next->next
```

(note the dotted lines)