Records

Records Within Records

This name is now a type which can be used anywhere a type such as "Num" can be used.

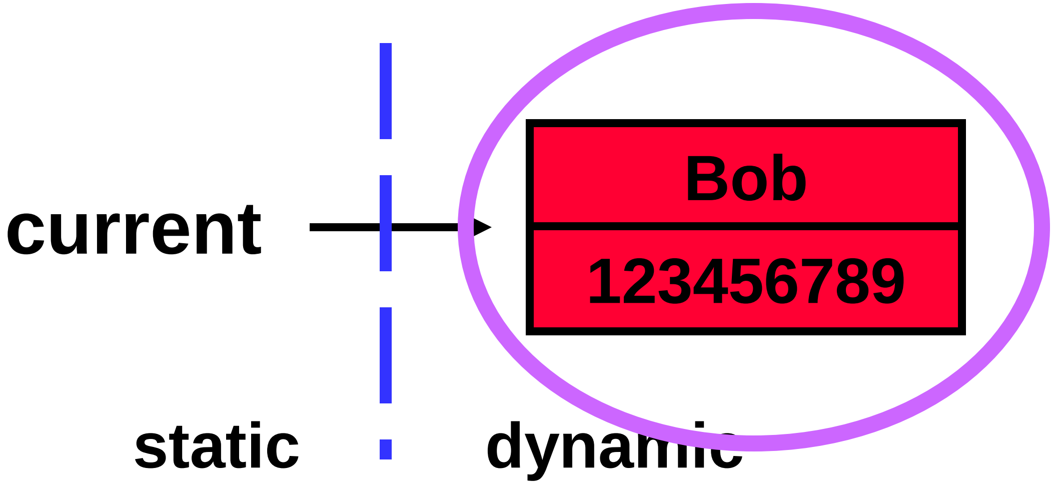
What are these called?

Date_Type

Date_Type

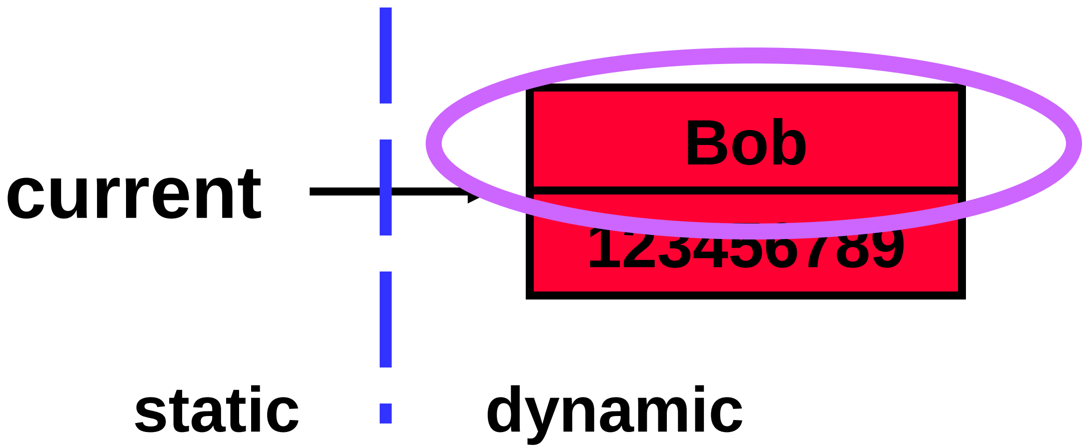
Types

Pointers and Records



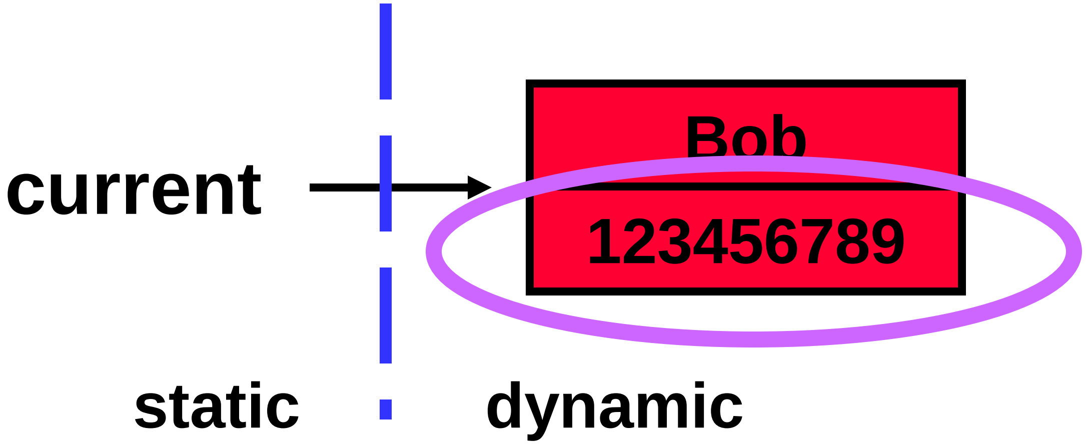
`current^`

POINTERS AND RECORDS




```
current^.name <- "Bob"
```

Pointers and Records




```
current^.SSN <- 123456789
```

What's the Big Deal

- **We already knew about static data**
- **Now we see we can allocate dynamic data but**
- **Each piece of dynamic data seems to need a pointer variable and pointers seem to be static**
- **So how can this give me flexibility**

LB

Questions?

Introduction to Linked Lists

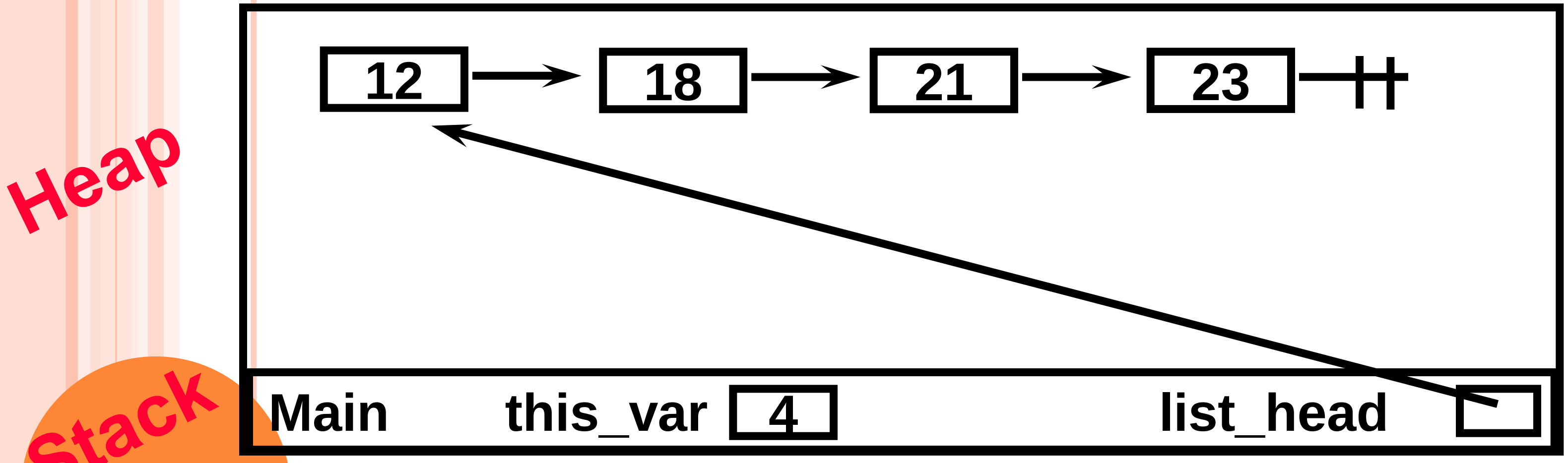
PROPERTIES OF LISTS

- **We must maintain a list of data**
- **Sometimes we want to use only a little memory:**

- **Sometimes we need to use more memory**

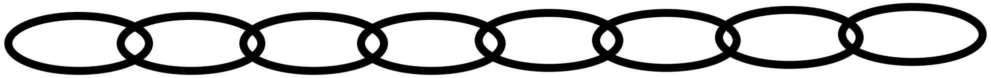
- **Declaring variables in the standard way won't work here because we don't know how many variables to declare**
- **We need a way to allocate and de-allocate data dynamically (i.e., on the fly)**

Linked Lists "Live" in the Heap

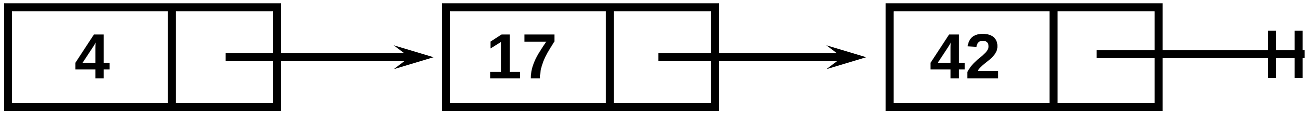


- The **heap** is memory not used by the **stack**
- **Dynamic** variables live in the **heap**
- We need a pointer variable to access our list in the heap

LINKED LISTS



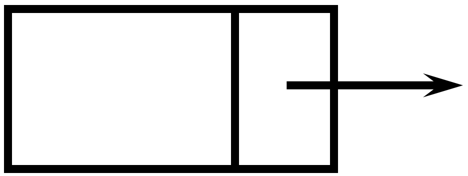
With pointers, we can form a "chain" of data structures:




```
List_Node definesa Record
     data isoftype Num
     next isoftype Ptr toa List_Node
endrecord  //List_Node
```

LINKED LIST RECORD TEMPLATE

```
<Type Name> definesa record
  data isoftype <type>
  next isoftype ptr toa <Type Name>
endrecord
```



Example:

```
  Char_Node definesa record
    data isoftype char
    next isoftype ptr toa Char_Node
  endrecord
```

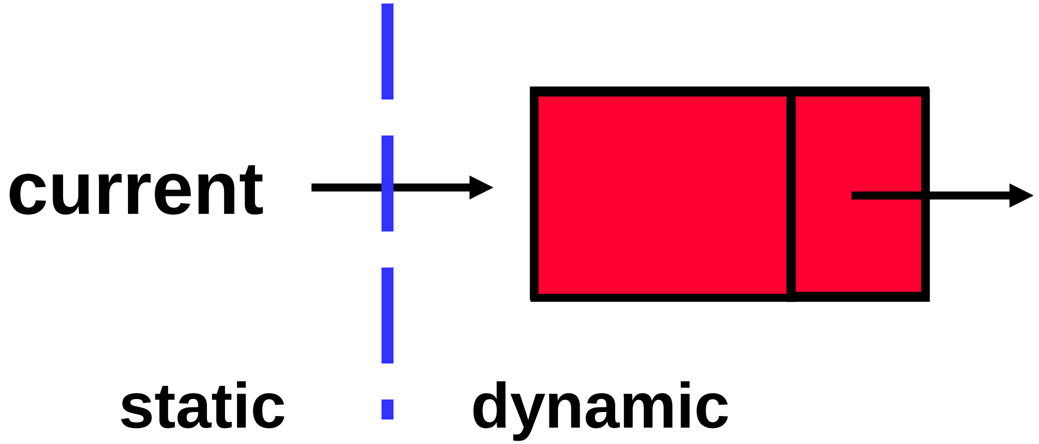
Creating a Linked List Node

```
Node definesa record
  data isoftype num
  next isoftype ptr toa Node
endrecord
```

And a pointer to a Node record:

```
current isoftype ptr toa Node
current <- new(Node)
```

Pointers and Linked Lists

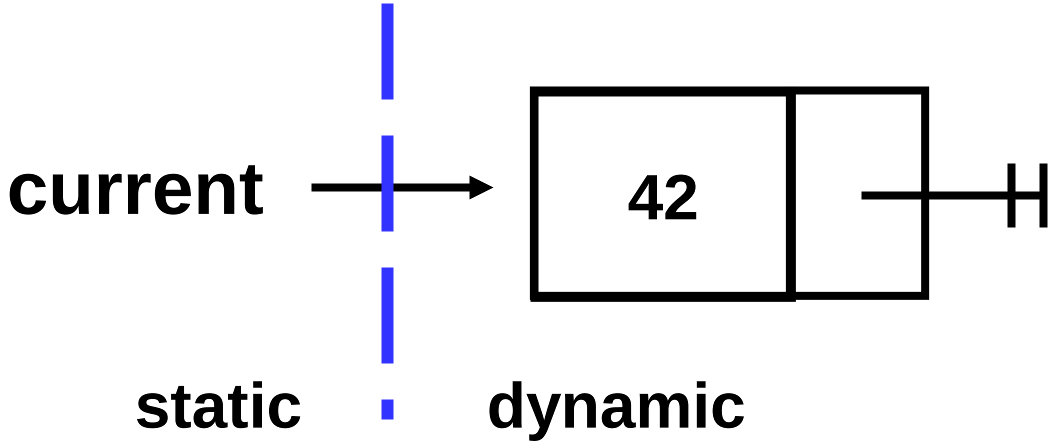



```
current^

current^.data

current^.next
```

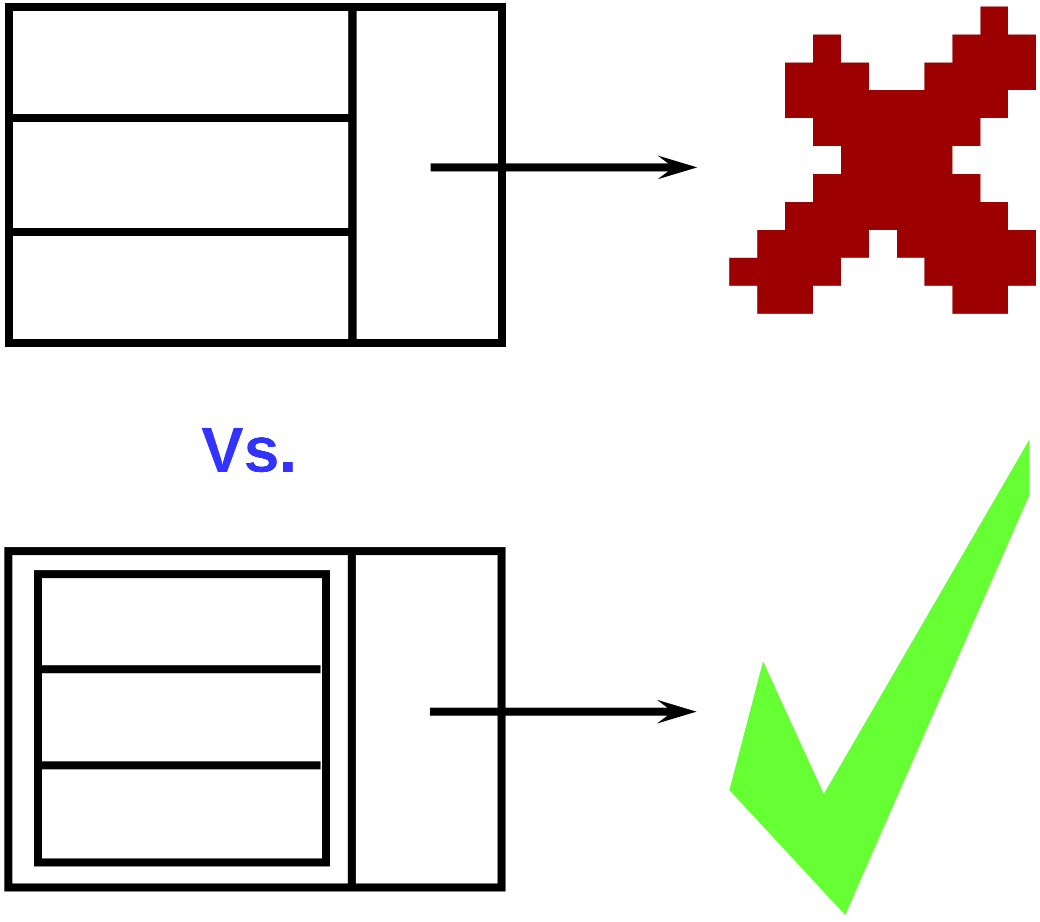
Accessing the Data Field of a Node




```
current^.data <- 42

current^.next <- NIL
```

Proper Data Abstraction



Complex Data Records and Lists

The examples so far have shown a single num variable as node data, but in reality there are usually more, as in:

```
Node_Rec_Type definesa record
   this_data isoftype Num
   that_data isoftype Char
   other_data isoftype Some_Rec_Type
   next isoftype Ptr toa Node_Rec_Type
endrecord // Node_Rec_Type
```

pda(5)

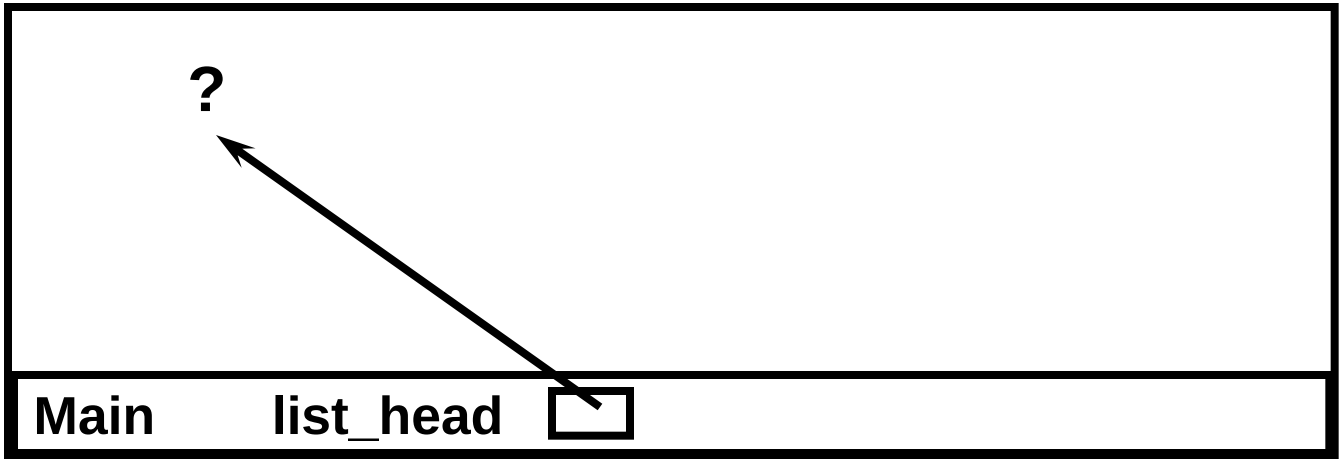
A Better Approach with Higher Abstraction

One should separate the data from the structure that holds the data, as in:

```
Node_Data_Type definesa Record
  this_data isoftype Num
  that_data isoftype Char
  other_data isoftype Some_Rec_Type
endrecord // Node_Data_Type


Node_Record_Type definesa Record
  data isoftype Node_Data_Type
  next isoftype Ptr toa Node_Rec_Type
endrecord  // Node_Record_Type
```

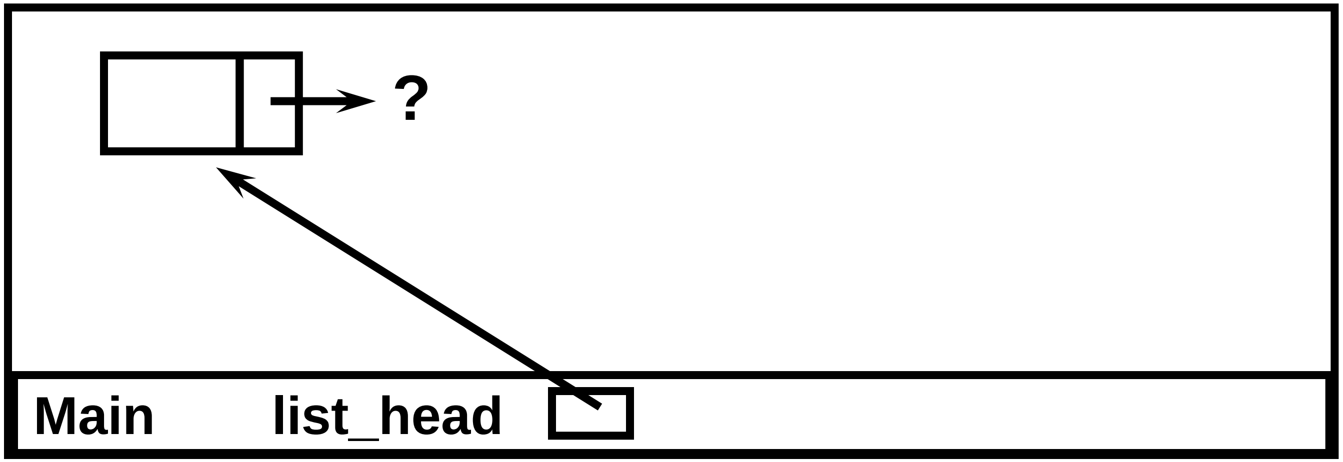
Creating a Pointer to the Heap




```
list_head isoftype ptr toa List_Node
```

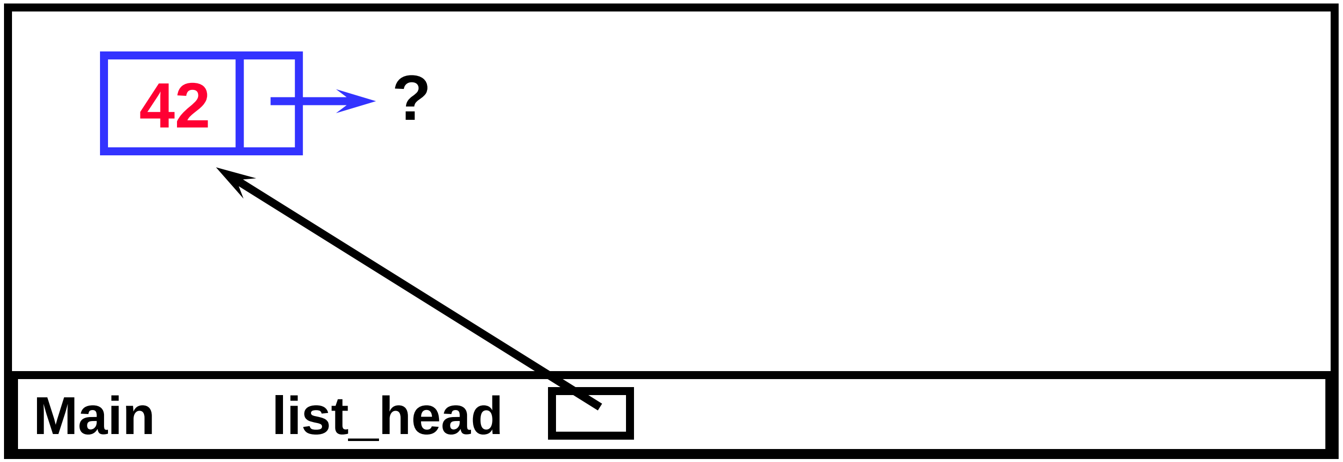
Notice that list_head is not initialized and points to "garbage."

Creating a New Node in the List




```
list_head <- new(List_Node)
```

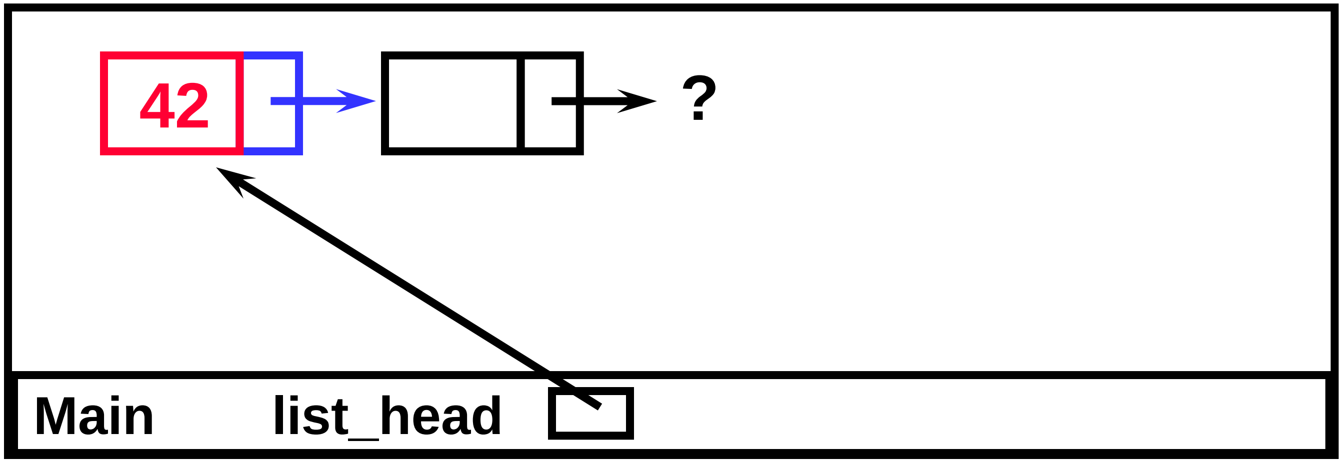
Filling in the Data Field




```
list_head^.data <- 42
```

The ^ operator follows the pointer into the heap.

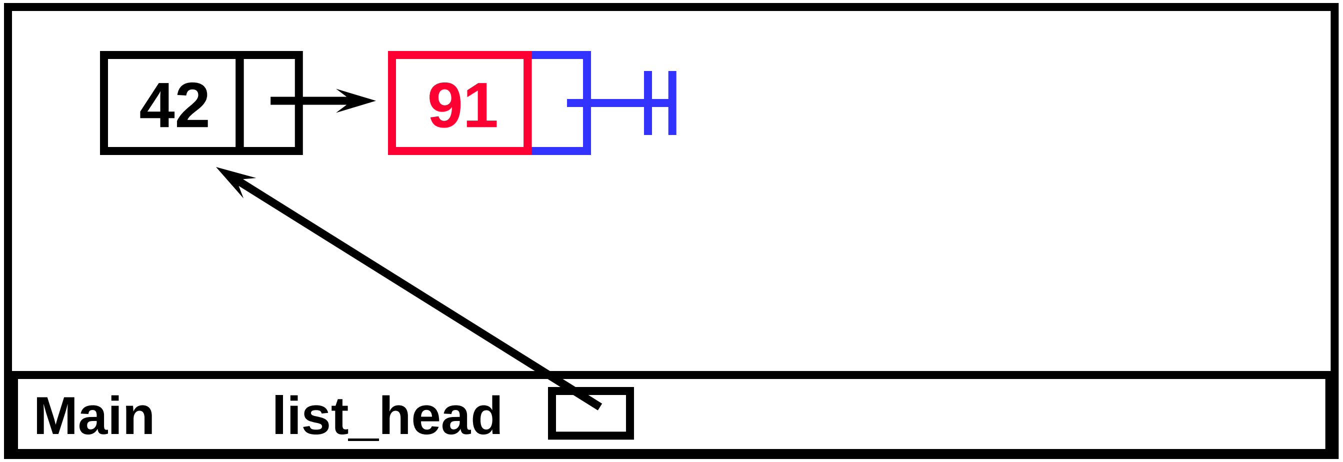
Creating a Second Node




```
list_head^.data <- 42
list_head^.next <- new(List_Node)
```

The "." operator accesses a field of the record.

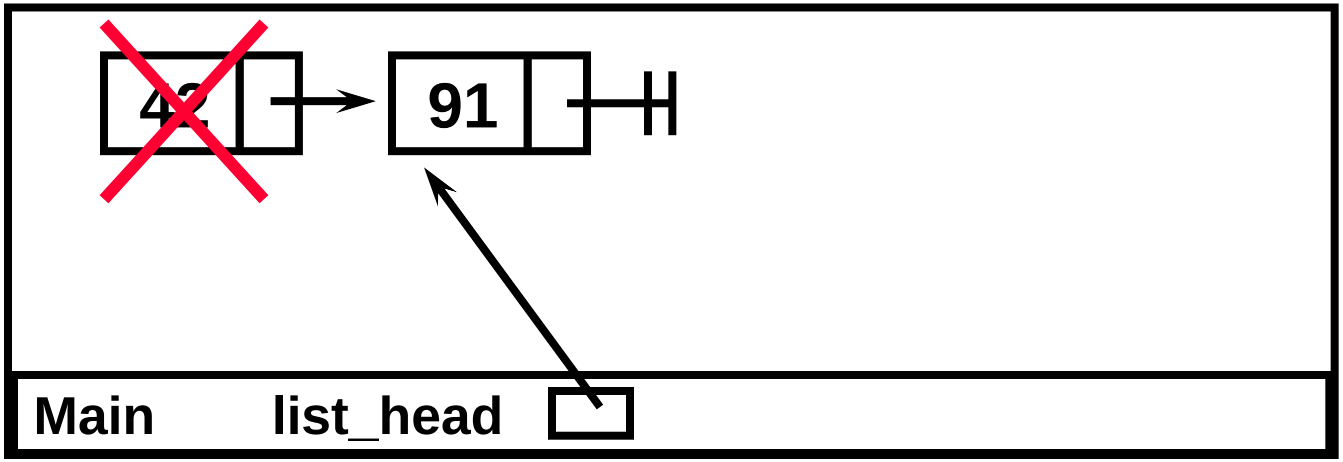
Cleanly Terminating the Linked List




```
list_head^.next^.data <- 91
list_head^.next^.next <- NIL
```

We terminate linked lists "cleanly" using NIL.

Deleting by Moving the Pointer



If there is nothing pointing to an area of memory in the heap, it is automatically deleted.

```
list_head <- list_head^.next
```