Massey University
COLLEGE OF SCIENCES

159201

Week 1

Data Types

- Basic Data Types
- Integers, real, characters, boolean ...
- e.g., in C/C++
- int
- float
- char
- bool (C has no boolean types, programmers use #define instead)

Data Types

- Basic Data Types **grouped together**
- *Structured Data Types:*
- *Arrays, strings, records*
- *In C/C++ we use* **`struct`**

```
struct BookRecord {
    char title[40];
    float callnumber;
};
```

Data Types

```
struct BookRecord {
   char title[40];
   float callnumber;
};
...
main(){
...
BookRecord book;
book.callnumber = 5.265;
}
```

Reference and pointers

Recall from 159101 / 159102:

`*` a pointer (declare a pointer to any type)

`new` allocates memory (equivalent to C malloc())

`&` the address of a variable.

`->` the element of a pointer to a structure

Examples with *

```
#include <stdio.h>
main(){
      int a=10;
      int *b;
      b=&a; //the address is the same
      printf("a=%d and b=%d \n",a,*b);
}
```

Result: a=10 and b=10

Examples with funct(type *&)

```c
#include <stdio.h>
int b;
void function1(int *a) { a=&b; }
void function2(int *&a) { a=&b; }
main(){
        int *a;
        int x=10;
        a=&x;
        printf("a=%d ",*a);
        b=20;
        function1(a);
        printf("a=%d ",*a);
        b=30;
        function2(a);
        printf("a=%d \n",*a);
}
```

Result: ? ? ?

Examples with funct(type *&)

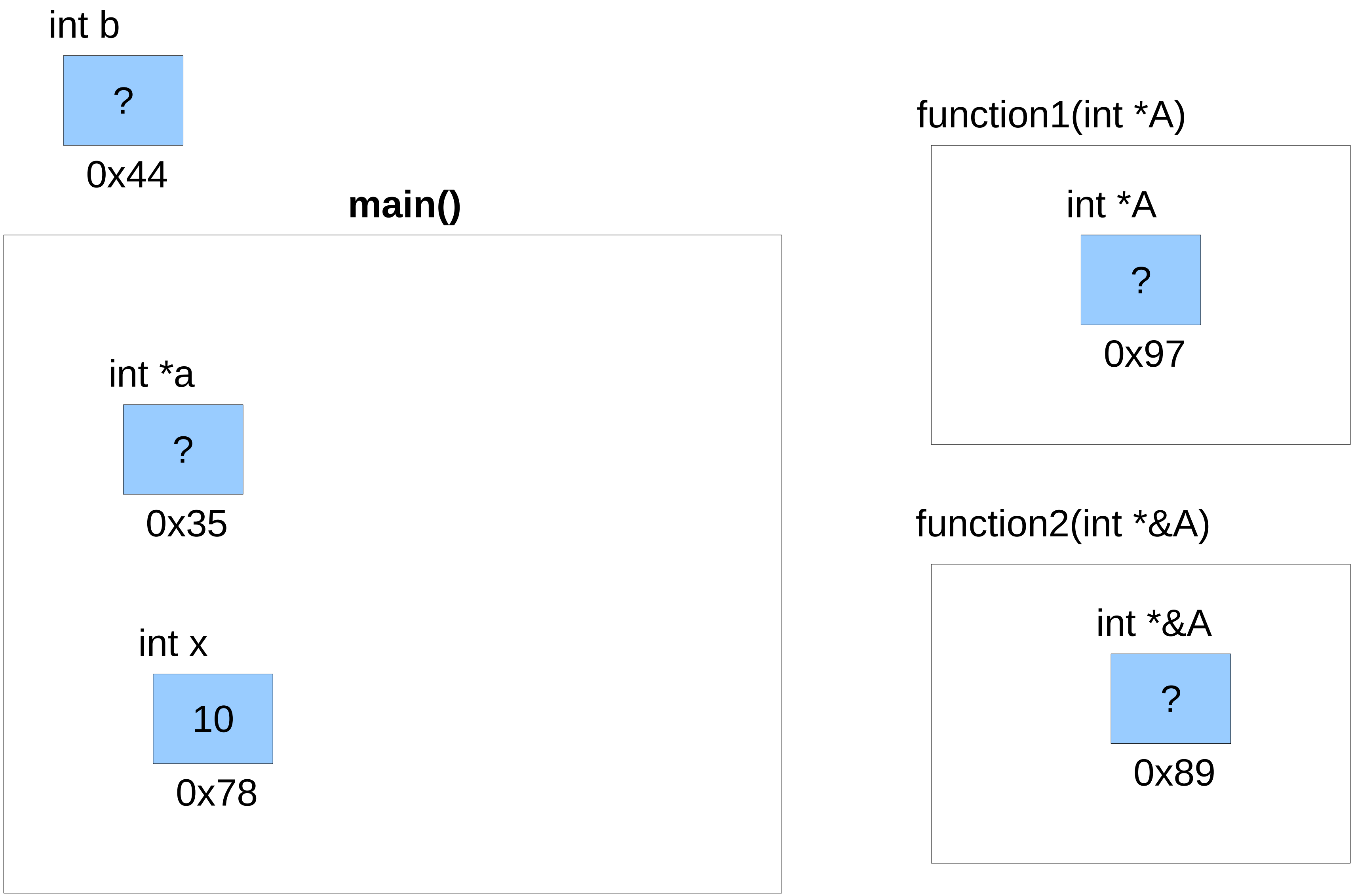
```c
#include <stdio.h>
int b;
void function1(int *a) { a=&b; }
void function2(int *&a) { a=&b; }
main(){
        int *a;
        int x=10;
        a=&x;
        printf("a=%d ",*a);
        b=20;
        function1(a);
        printf("a=%d ",*a);
        b=30;
        function2(a);
        printf("a=%d \n",*a);
}
```

Result is: 10 10 30

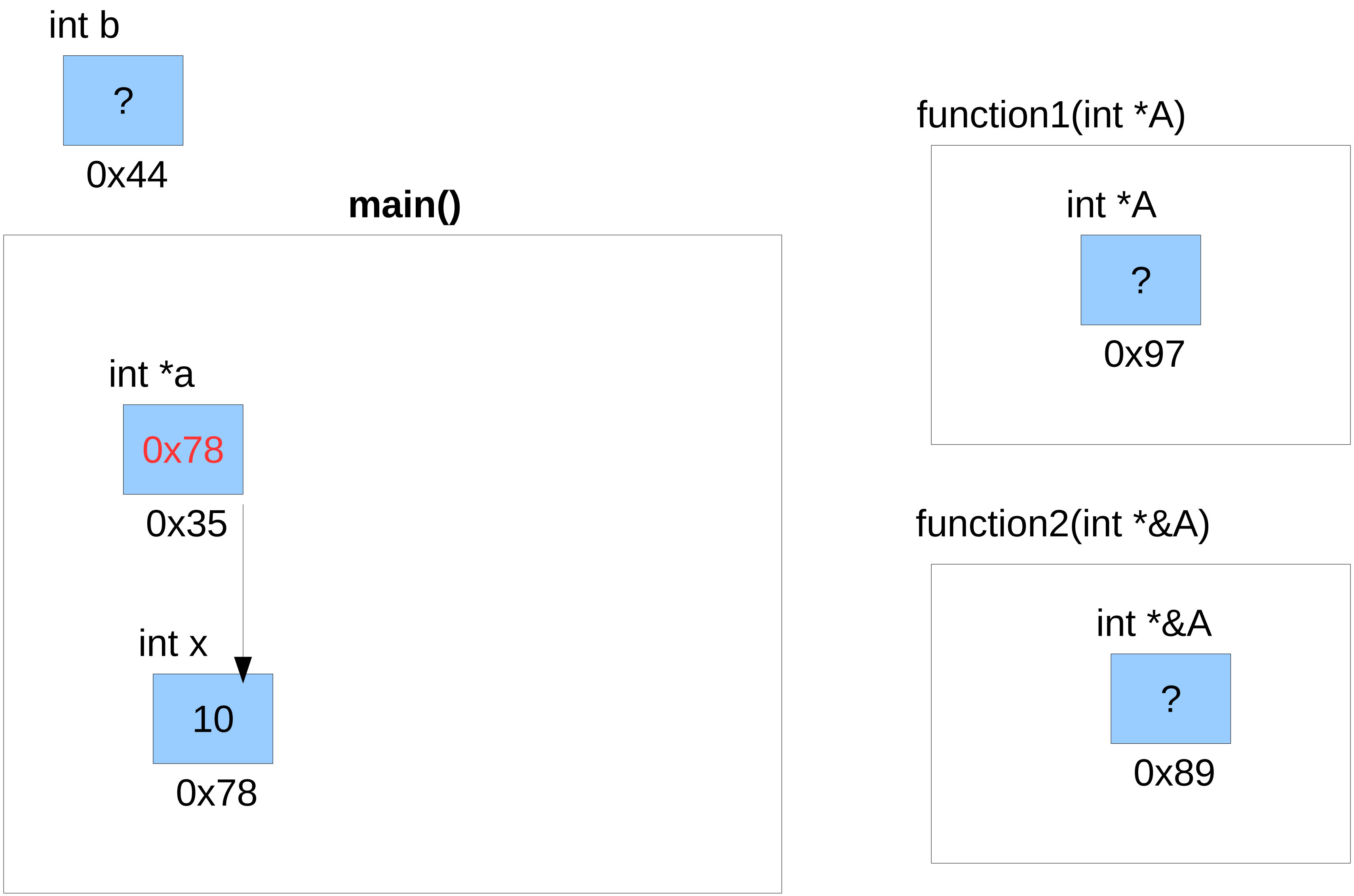
NOT: 10 20 30

Massey University
COLLEGE OF SCIENCES

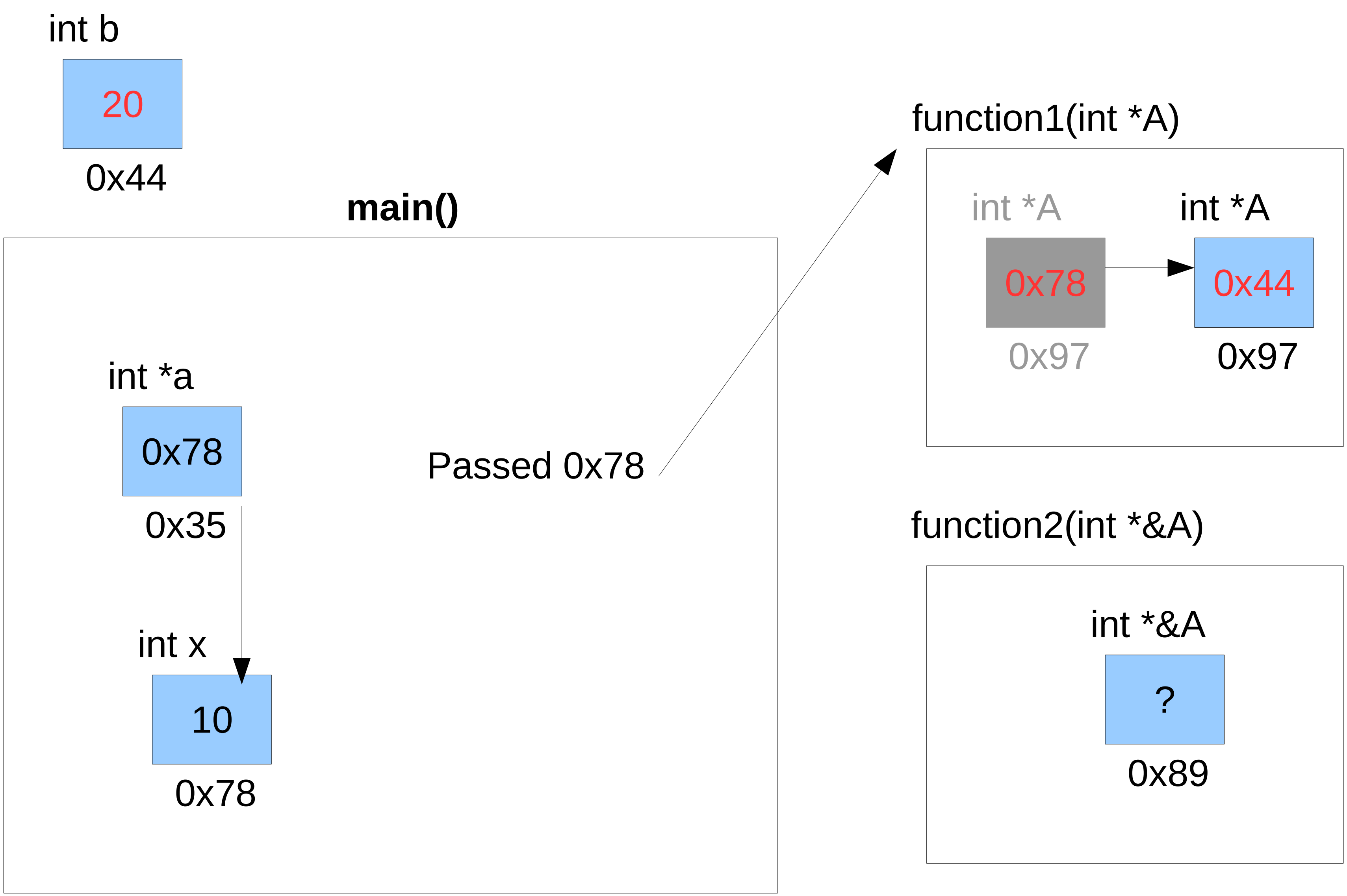
State of memory at line 7



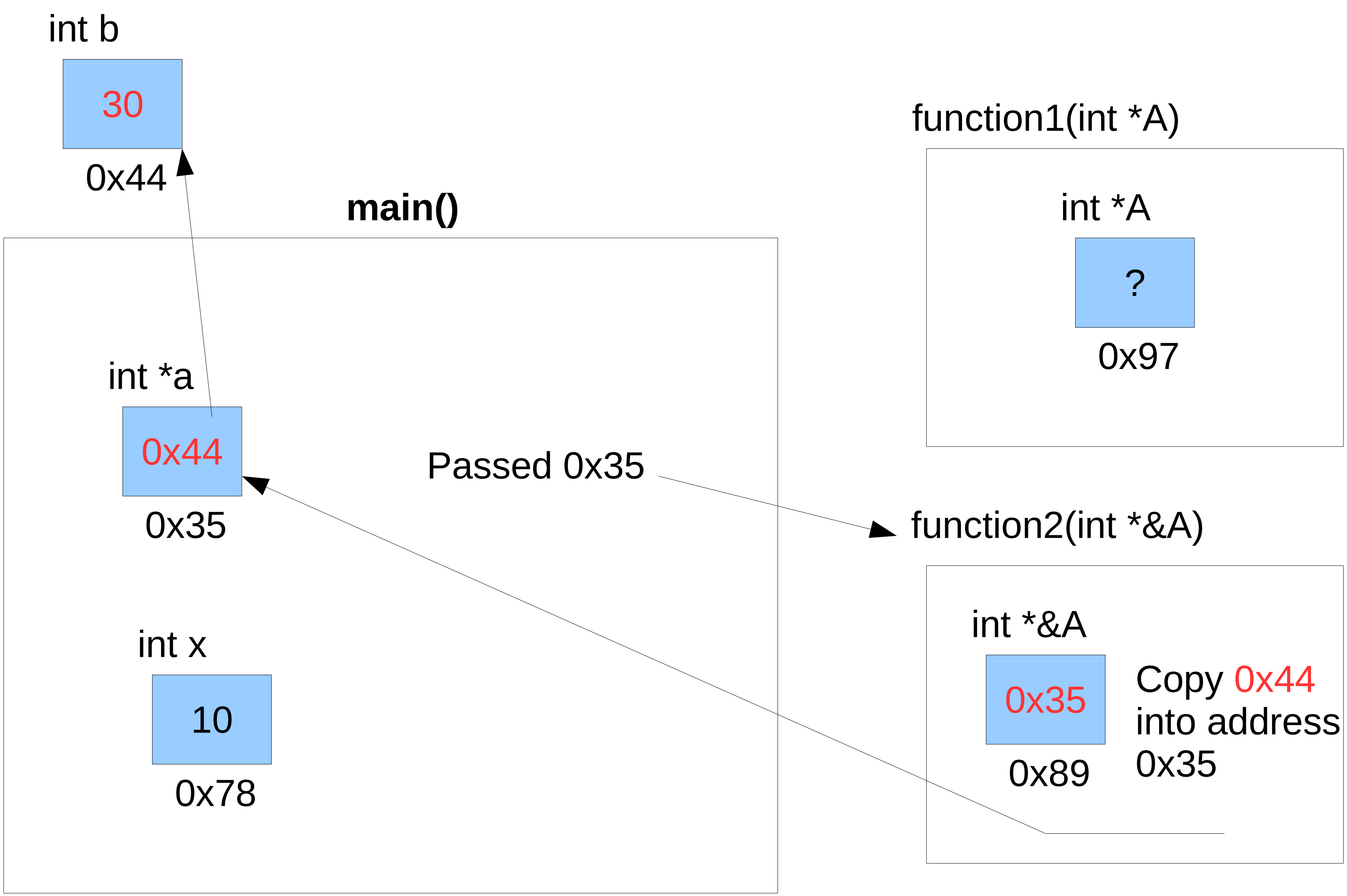
State of memory at line 9



State of memory at line 12



State of memory at line 15



Examples with funct(type *&)

The trick is to pass a pointer to a pointer...

This can be done passing *& (typical for C++) or

** (in C).

Run the program `code1_alternative.cpp` and play with the different variables. Try to follow what is happening to the addresses within the pointers.

malloc() and free(), New, delete

In C, memory allocation/deallocation:

Malloc() and free()

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

main(){
        int a[10];//static, 10 places
        int *b;//pointer only, no allocation yet
        b=(int*) malloc(10*sizeof(int));
        a[5]=10;
        b[5]=10;
        printf("result: a=%d and b=%d\n",a[5],b[5]);
        free(b);
}
```

NOTE: using unallocated pointers or freeing twice leads to disaster... (segmentation fault)

malloc() and free(), New, delete

In C++, memory allocation/deallocation:

New and **delete**

```
#include <stdio.h>

main(){
	int a[10];//static, 10 places
	int *b;//pointer only, no allocation yet
	b = new int[10];//allocate
	a[5]=10;
	b[5]=10;
	printf("result: a=%d and b=%d\n",a[5],b[5]);
	delete[] b;//deallocate (use object's destructor)
}
```

NOTE: new and delete have specific roles in OO (constructors and destructors), more in 159234

What -> means?

Remember that "." is used to refer to elements of structures, e.g.

```
book.callnumber
```

However, when "book" is a pointer we have to refer to it using "->", e.g.

```
BookRecord book;
BookRecord *bookpointer;
...
main(){...
    book.callnumber=10;
    bookpointer->callnumber=10;
}
```

Massey University
COLLEGE OF SCIENCES

Review: command line args

Remember how to get arguments from command line?

```
int main(int argc, char **argv) {
   if(argc!=3) {
     cout << "needs: Q1phase Q2phase" << endl;
     exit(0);
   }
   Q1phase=atoi(argv[1]);
   Q2phase=atoi(argv[2]);
…
}
```

Review: command line args

When the operating system receives the command to run an executable, it reads an expression. This expression can be split into several strings:

```
#myprog.exe argument1 argument2 argument3 …

 argv[0]     argv[1]   argv[2]   argv[3] …
```

`argc:` holds the number of arguments (4 in this case)

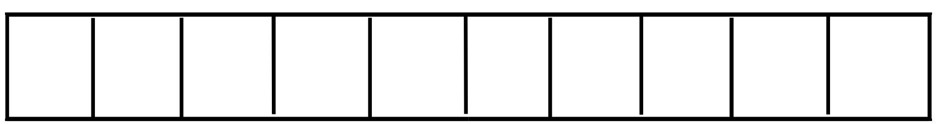
Abstract Data Types (ADT)

- ADT: a ***model*** for ***data structure*** with a certain behaviour and certain ***properties***.
- Advantages of ADTs
  - Reduce details – allow focus to be on the “main picture”.
  - Different implementations can be used.
  - Underlying implementation can be changed or upgraded.
  - In C++, it is convenient to implement an ADT as a **class.**

Revision of Arrays

- Remember arrays in C or C++? Example:
- int x[10]; // ten elements x[0], x[1] ... x[9]

0 1 2 3 4 5 6 7 8 9 These are the **index** numbers



Revision of Arrays

Advantages of Arrays

Simple, Fast, Random access

Disadvantages of Arrays

Fixed size – too small or too big at runtime

Difficult to insert or delete without leaving spaces

e.g., we have 10 elements, delete array[2] and array[7].

2D arrays

Example:

```
int matrix[4][4];
```

At some point, matrix`[2][2]=64;`

	0	1	2	3
0				
1				
2			64	
3				

2D arrays

- 2D Arrays in C/C++ are stored as a 1D array
- **Row-major** order
- Known in math as a **matrix**
- **Sparse matrix** has few numbers and lots of elements with value = 0

Row-major X colum-major

Row-major order? How do we know?

```
#include <stdio.h>
int a; int b;
int matrix[4][4];
main(){
  for(a=0;a<4;a++){
    for(b=0;b<4;b++){
      printf("%p ",&matrix[a][b]);//pointers
    }
    printf("\n");
  }
}
```

Row-major X colum-major

Output:

6293920 6293924 6293928 6293932

6293936 6293940 6293944 6293948

6293952 6293956 6293960 6293964

6293968 6293972 6293976 6293980

- the output may not the same for different machines, even for different runs.

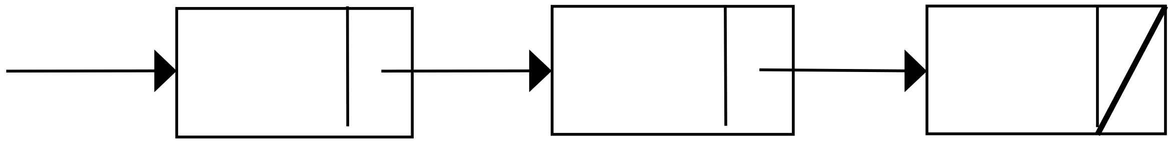
- However, it follows a pattern: a space of **4 (bytes)** between elements within the same row.

- The first element in the second column is +4 bytes from the last element in the first row

→ row-major confirmed

Linked-lists

Lets study our first ADT...



Linked-lists

Linked-lists are ***sequences of connected nodes***.

Linked-lists are empty at the start.

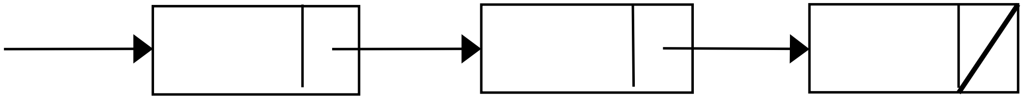
Nodes are added ***dynamically*** (at runtime).

Nodes contain pointers to other nodes.

The address of the list is the pointer to the first node.

Linked-lists can be used as an alternative to arrays.

Linked-lists are ***linear*** structures, i.e., one element points to another single element.



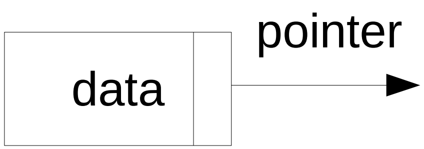
Linked-lists compared to arrays

Linked-lists	Arrays
Grows during runtime	Fixed size (compilation time)
Dynamic memory allocation	Static memory allocation
Easy to insert/delete in the middle	Inserting elements leave empty spaces in memory
Sequential access is fast	Random access (index)
slow	fast
Complicated (needs extra functions to work)	Simple

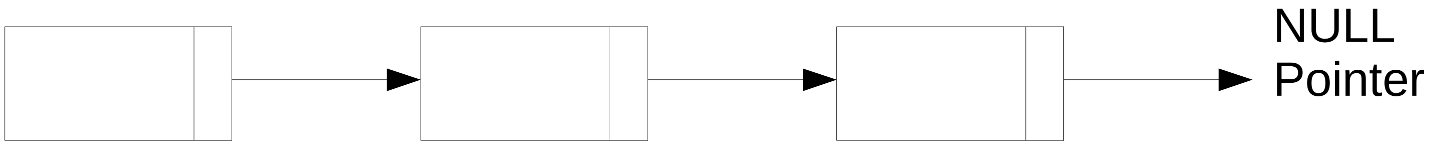
Linked-lists

We can represent linked-lists schematically:

A ***node*** is represented like this:

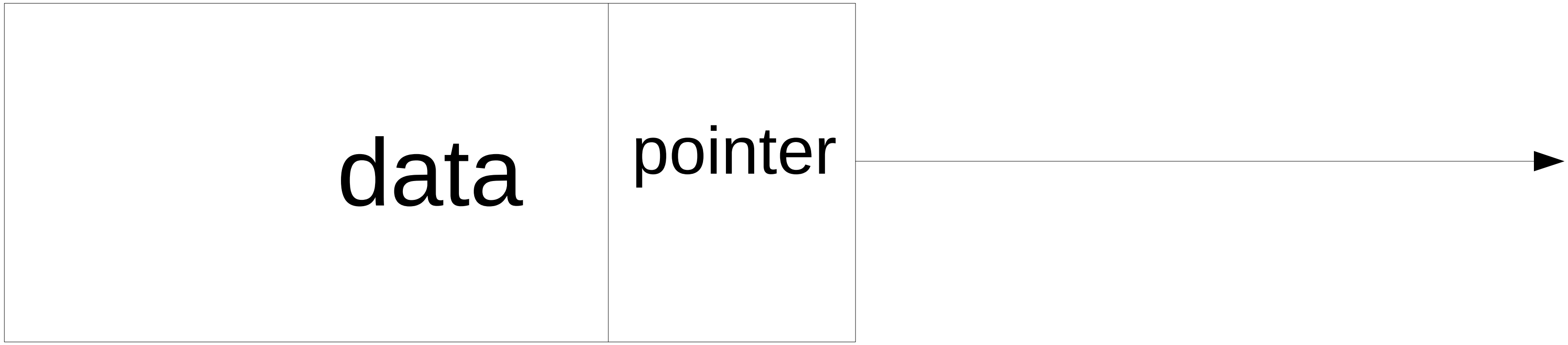


A ***set of nodes*** composes a linked-list:



Linked-lists

Each ***node*** contains *data* and a *single pointer*:

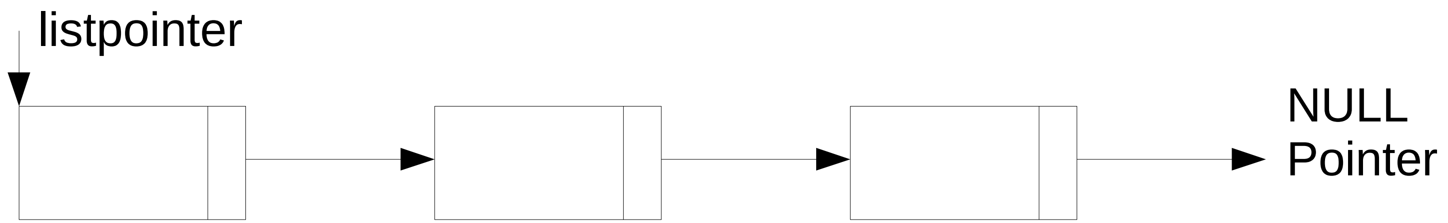


Linked-lists

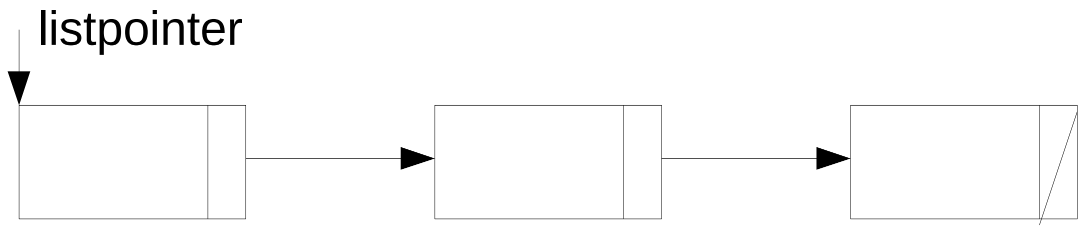
The linked-list can have (theoretically) an ***infinite*** number of node (or elements). How do we know when the list begins or ends?

The linked-list ***begins*** with a pointer: e.g., listpointer

The linked-list ***ends*** with a NULL pointer.



Alternatively, represented schematically by:



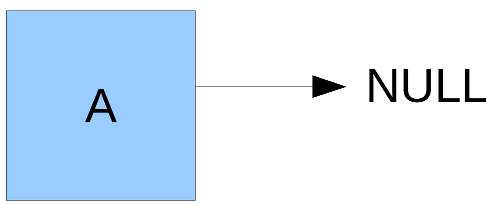
Linked-lists: implementation

```
struct Node {  //declaration of a Node
   int accnumber; //some data...
   float balance; //more data...
   struct Node *next; //the pointer to the next node
}; //Note the recursive declaration
typedef struct Node Node;//only for C
```

Linked-lists

Until one declares a Node and specifically allocates memory to it, no memory is allocated:

```
Node *A; //declare one pointer to a linked-list called 'A'
A = NULL;
```



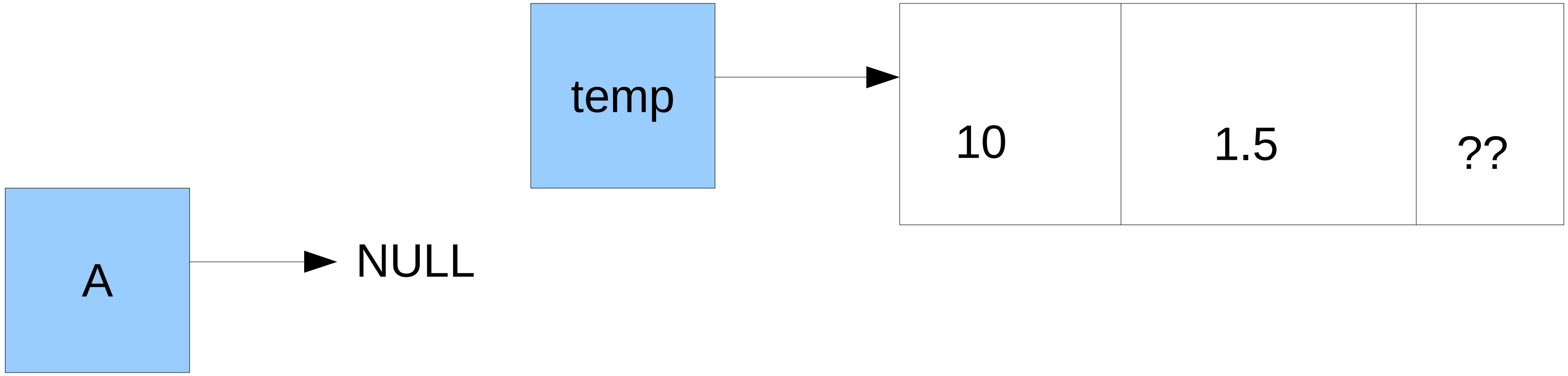
REMEMBER: there is no place for an int or a float yet...

There is only a pointer to a Node, no allocated memory.

Linked-lists

Lets add, *manually* (at main()), a new node on list **A:**

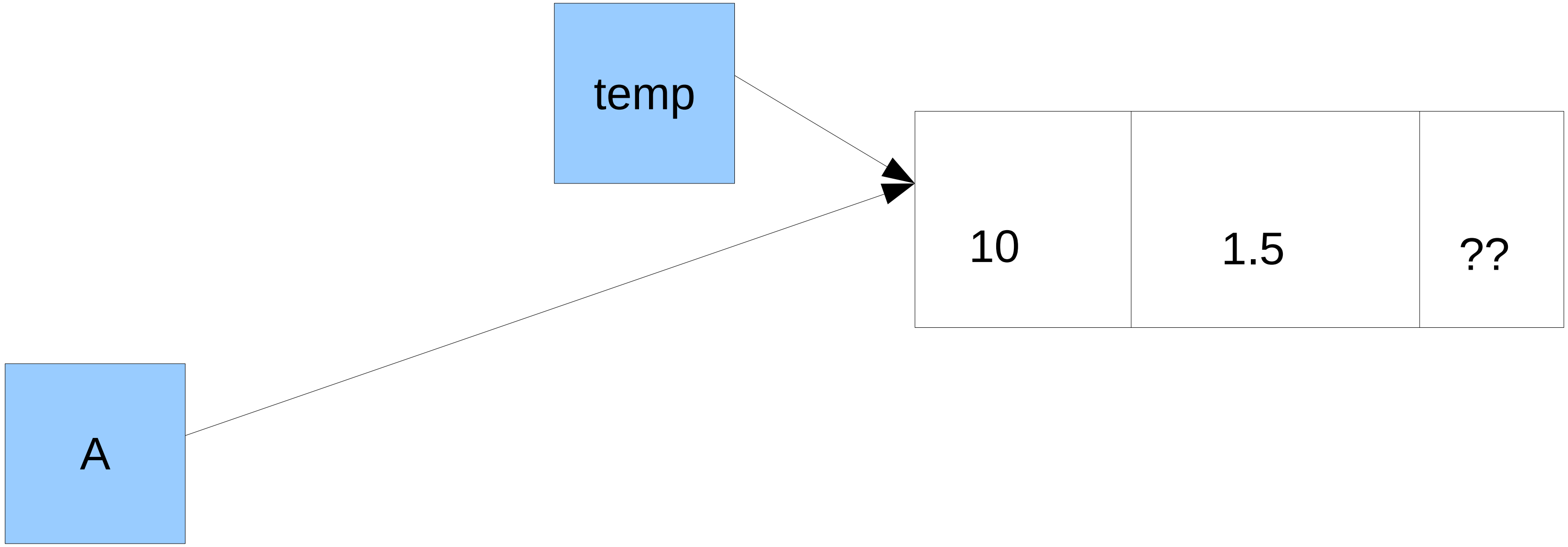
```
Node *temp; //declare a temporary pointer to a Node
temp = new Node; //allocate space
temp->accnumber=10; //load the value
temp->balance=1.5;//load the value
```



Linked-lists

The new element should be pointed by A. We can copy the content of temp to A:

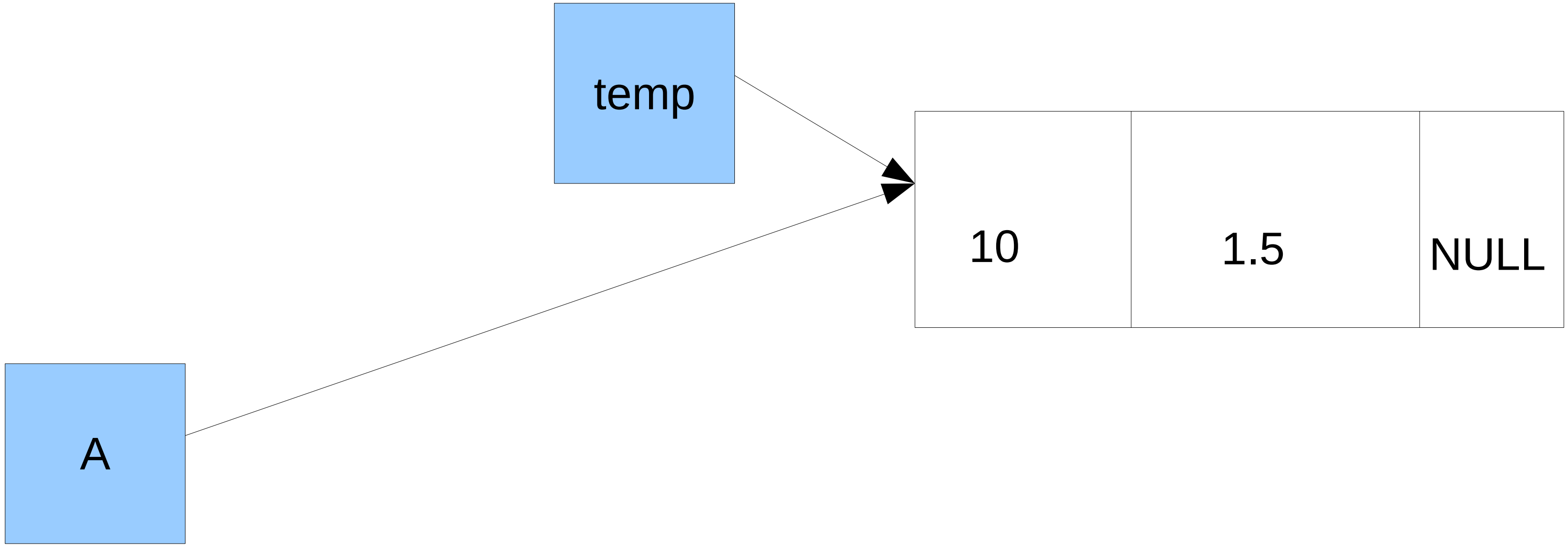
```
A = temp;
```



Linked-lists

A has now one element. But the new element ***`next`*** pointer points to a **random** place in memory. Lets point it to NULL:

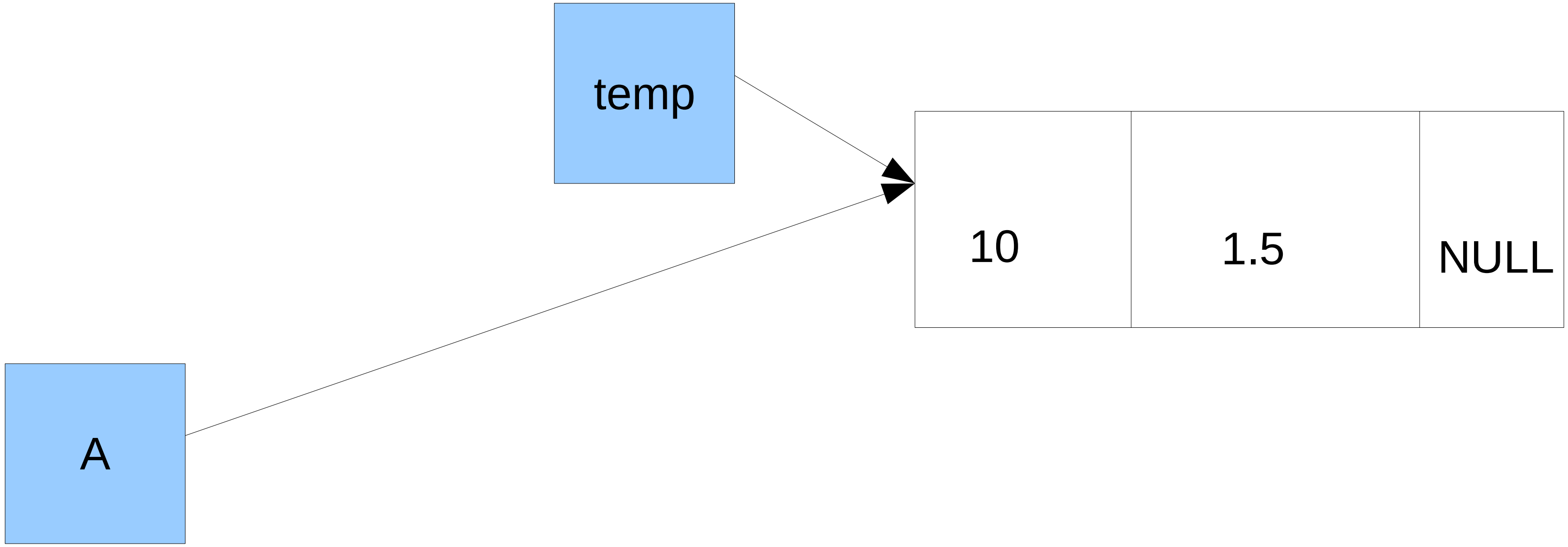
```
temp->next=NULL; //(or A->next=NULL)
```



Linked-lists

Now, if we want to refer to the ***first*** element of A:

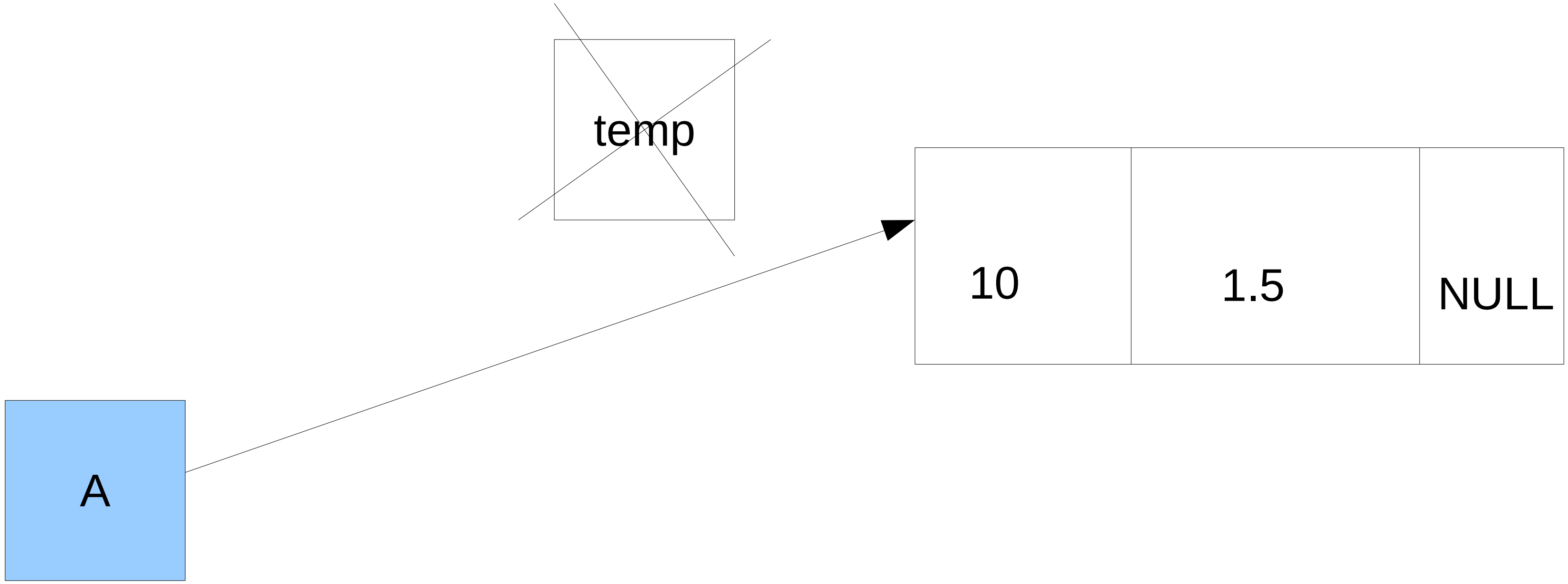
```
A->accnumber
A->balance
A->next ...
```



Linked-lists

We could now eliminate temp. (we will see how to do this properly inside a function)

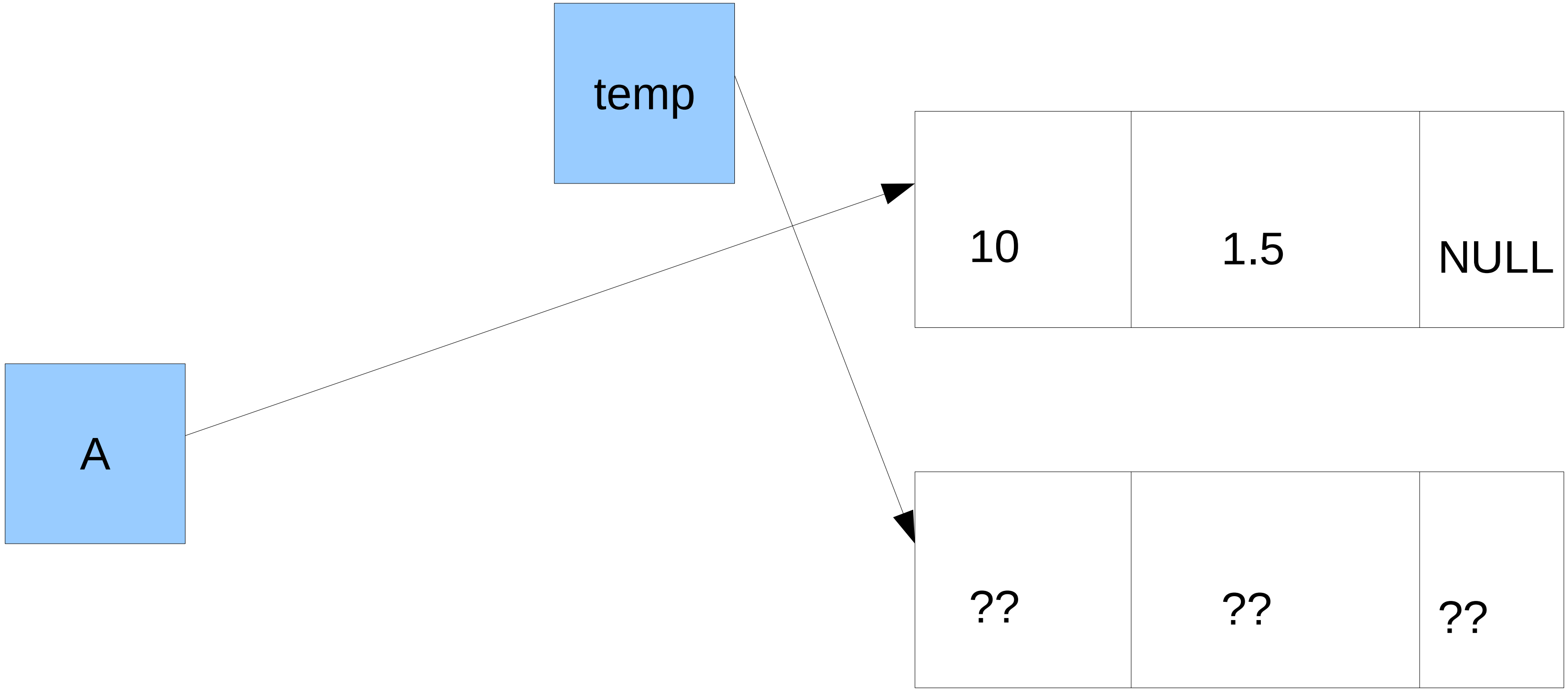
But why do we need temp in the first place? Lets use temp to create a second element instead of deleting it...



Linked-lists

Suppose you want a second element linked to list A.

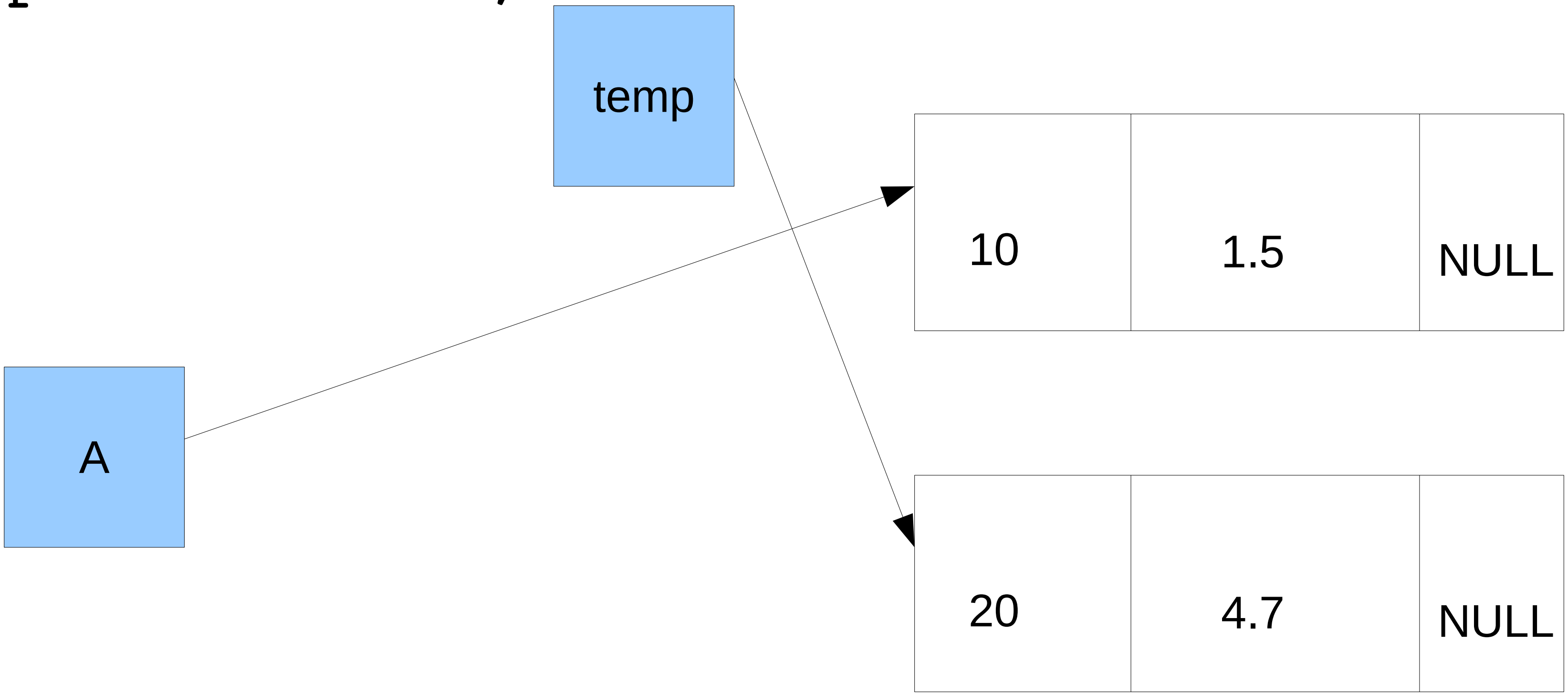
```
temp = new Node;//used the same pointer, but this points to a new location in memory
```



Linked-lists

Load the second set of values into it:

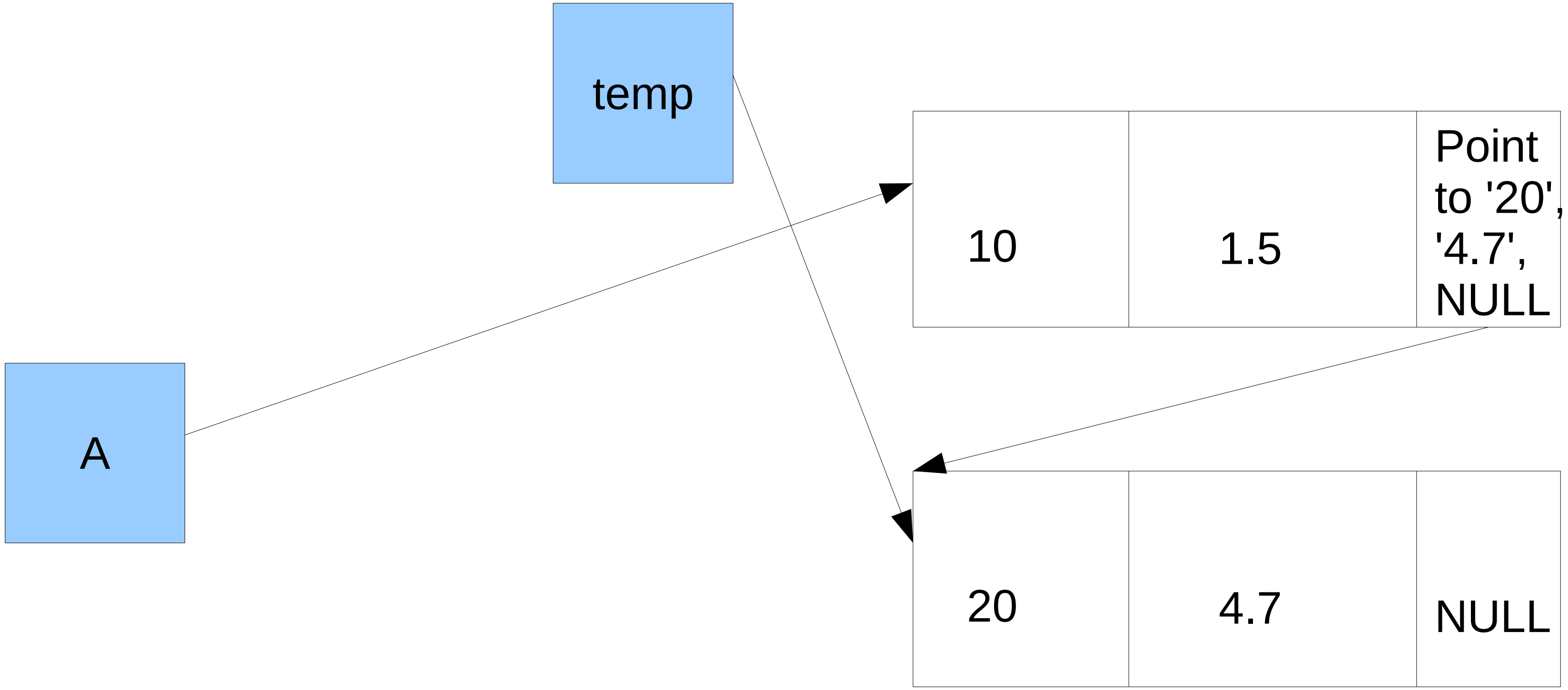
```
temp->accnumber=20;//load the values
temp->balance=4.7;
temp->next=NULL;
```



Linked-lists

Then, link the *second* element to the *first*:

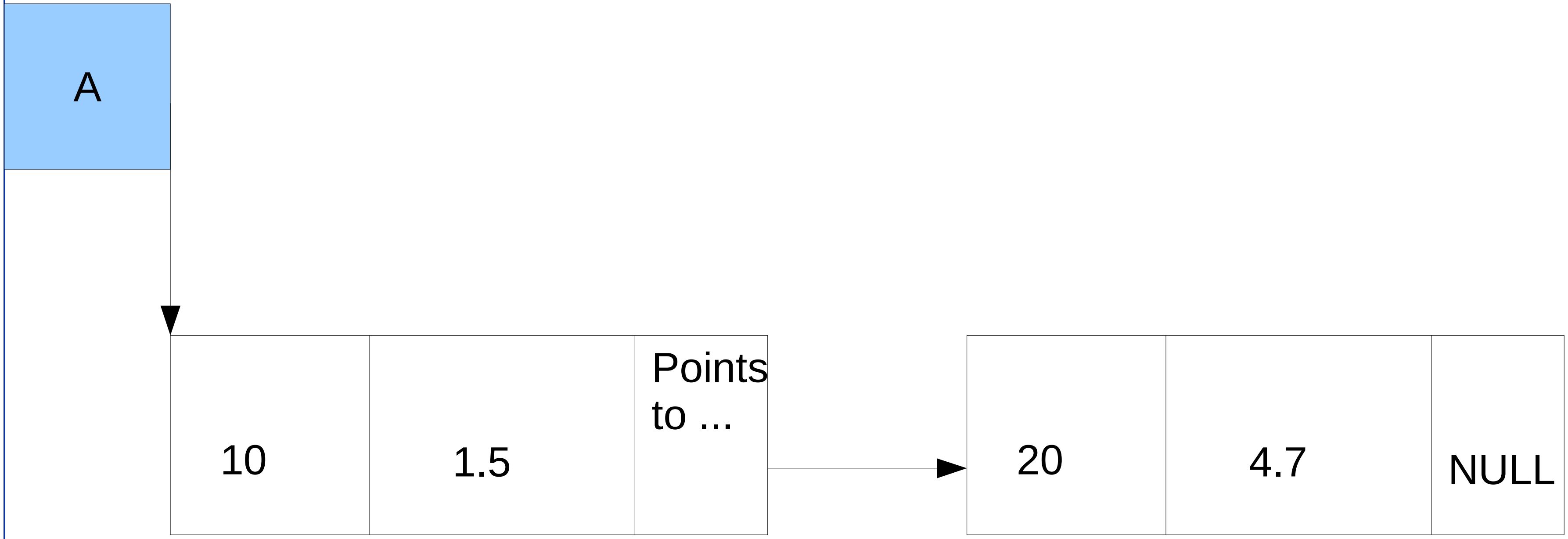
```
A->next=temp; //copies the location of temp
```



Massey University
COLLEGE OF SCIENCES

Linked-lists

Rearranging the figure, this is the linked-list at this point...



Discussion: what happens if we create more elements?

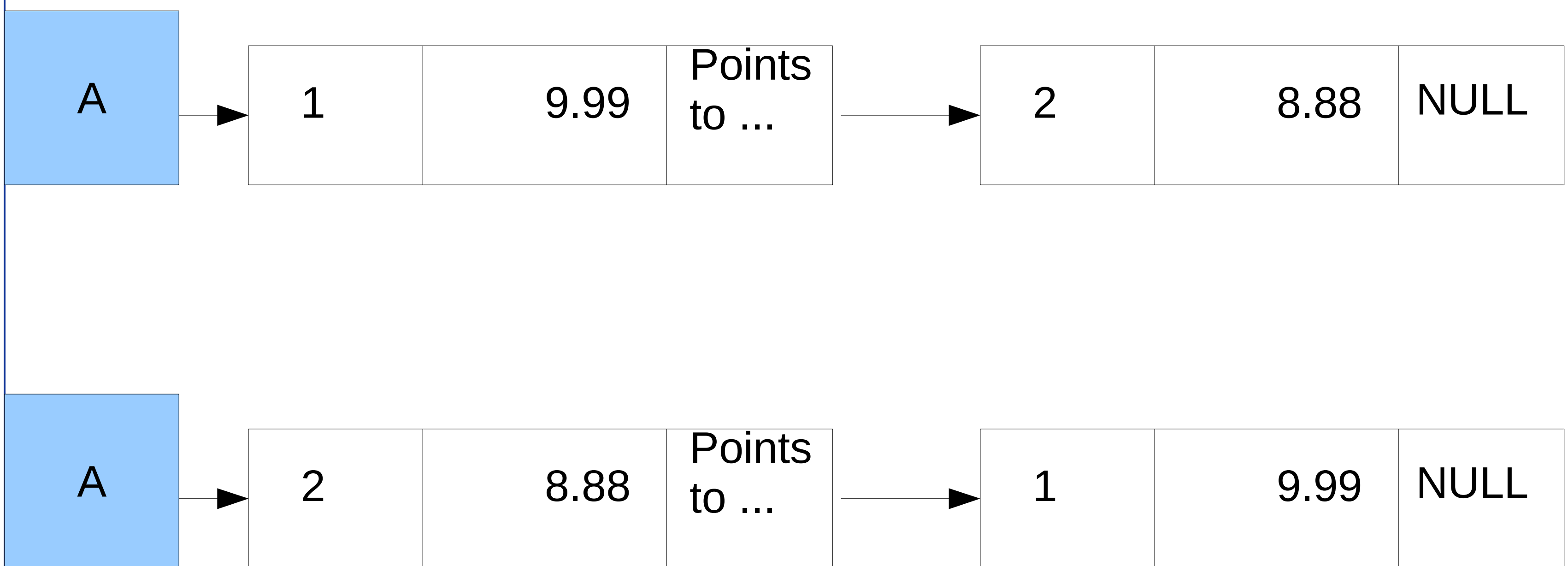
Add to the Head or to the Tail? Which one is simpler?

Sample Linked-list in C++

```
#include <stdio.h>
struct Node {  //declaration
        int accnumber;
        float balance;
        Node *next;
};
Node *A;  //declaration

int main() {
  A = NULL;
  AddNode(A, 1, 9.99);
  AddNode(A, 2, 8.88);
  AddNode(A,...);//other elements...
}
```

Head or Tail?



NOTES: Until we look inside AddNode(), we do not know in which ***order*** the elements will fit.

Two options: new elements *before* the Head, OR new elements *after* the Tail

AddNode() Algorithm

Pseudo-code: to explain the logic in “false” computer language

Algorithm AddNode
Required: List listpointer, integer a, float b
Output: void
1: create a Node temp
2: allocate memory to temp
3: copy a to accnumber of temp
4: copy b to balance of temp
5: copy listpointer to next of temp
6: copy temp to listpointer

Massey University
COLLEGE OF SCIENCES

AddNode() in C++

```
void AddNode(Node * & listpointer, int a, float b) {
// add a new node to the FRONT of the list
   Node *temp;
   temp = new Node;
   temp->accnumber = a;
   temp->balance = b;
   temp->next = listpointer;
   listpointer = temp;
}
```

Reference and pointers

Subtle syntax in C/C++

A function can get parameters using pointers and/or references:

```
void function1( Node * listpointer...
```

In this case, the pointer to listpointer is passed as reference (a copy of the address is made). Changing listpointer *does not alter A or B*

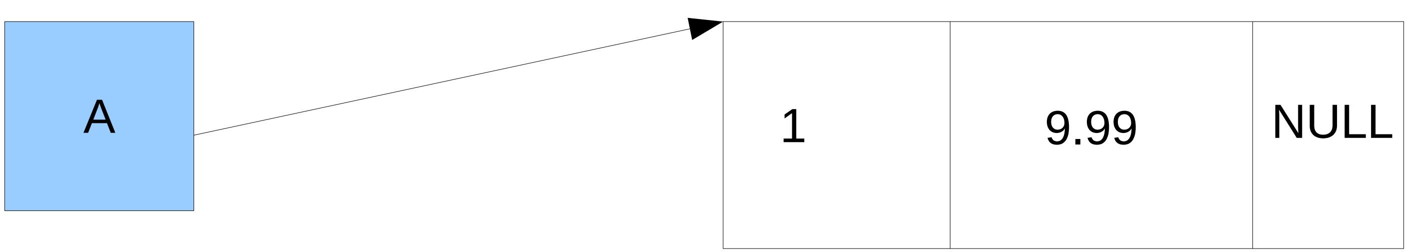
```
void function2( Node * &listpointer...
```

In this case, the pointer is itself passed to the function, so changing listpointer *changes A or B...*

AddNode(): where?

```
void AddNode(Node * & listpointer, int a, float b) {
// add a new node to the FRONT of the list
  Node *temp;
  temp = new Node;
  temp->accnumber = a;
  temp->balance = b;
  temp->next = listpointer;
  listpointer = temp;
}
```

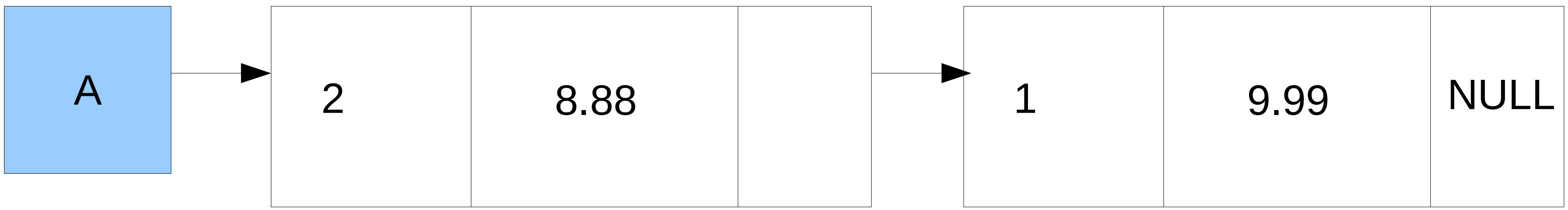
```
        AddNode(listpointer, 1, 9.99);
NOTE:   we start with A=NULL;
        Then, temp->next = NULL;
```



AddNode(): add 2nd node?

```
void AddNode(Node * & listpointer, int a, float b) {
// add a new node to the FRONT of the list
  Node *temp;
  temp = new Node;
  temp->accnumber = a;
  temp->balance = b;
  temp->next = listpointer;
  listpointer = temp;
}
```

```
        AddNode(listpointer, 2, 8.88);
NOTE:   we start with A pointing to '1';
        Then, temp->next points to '1'
        A points to temp (now '2')
```



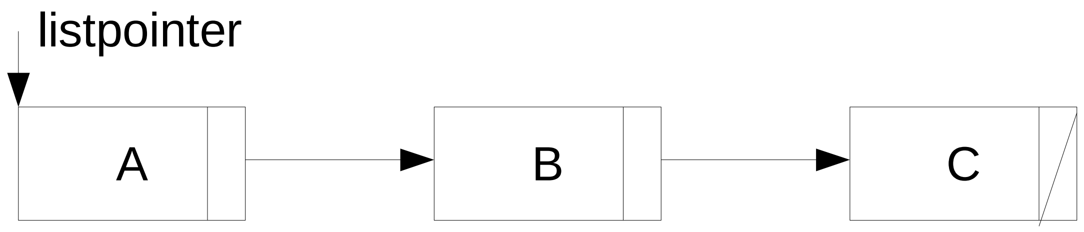
AddNode(): inserts to front

```
void AddNode(Node * & listpointer, int a, float b) {
// add a new node to the FRONT of the list
  Node *temp;
  temp = new Node;
  temp->accnumber = a;
  temp->balance = b;
  temp->next = listpointer;
  listpointer = temp;
}
```

Therefore, this AddNode() function inserts nodes to the **front** of the linked-list. The order is **REVERSED!**

PrintLL(): Printing a linked-list

- Now we want to print the linked-list
- However, we only have one pointer!
- How can we find all the other elements??
- We have to follow the **`next`** pointers....
- Schematically, we find the first node, then follow the pointers.



Print: step-by-step

- Create a pointer "`current`", of same type as node
- `current` initially points to the first element of the linked-list, `listpointer`
- At any point, if `current` is NULL, then we reached the ***end*** of the list (last element)
- While the loop condition is valid, we can print the contents of the element (node) of the linked-list:

```
current->accnumber
current->balance
```

Massey University
COLLEGE OF SCIENCES

Print the linked-list: pseudo-code

```
Algorithm PrintLinkedList
Required: List listpointer
Output: void
1: declare current (a pointer of type Node)
2: copy listpointer to current
3: while (true) do
4:    if (current is  NULL) then
5:      break
6:    end if
7:    print accnumber and balance
8:    copy next of current to current
9: end while
```

Notes: the key to understand the code is in line 8. The pointer called **`current`** moves one position forward.

Print the linked-list: pseudo-code

```
Algorithm PrintLinkedList
Required: List listpointer
Output: void
1: declare current (a pointer of type Node)
2: current = listpointer
3: while (true) do
4:    if (current == NULL) then
5:       break
6:    end if
7:    print accnumber and balance
8:    current = current->next
9: end while
```

Notes: the key to understand the code is in line 8. The pointer called **`current`** moves one position forward.

Print the linked-list

```
void PrintLinkedList(Node *listpointer) {
// print all elements
Node *current;
  current = listpointer;
  while (true) {
    if (current == NULL) { break; }
    printf("Account %i balance is %1.2f\n",
      current->accnumber, current->balance);
    current = current->next;//this is important!!
  }
  printf("End of the list.\n");
}
```

Observe how the **current** pointer is being used.
The infinite **while** loop can be modified later...

Print example

```
int main() {
   AddNode(A, 1, 9.99);
   AddNode(A, 2, 8.88);
   AddNode(A, 3, 7.77);
   PrintLinkedList(A);
}
```

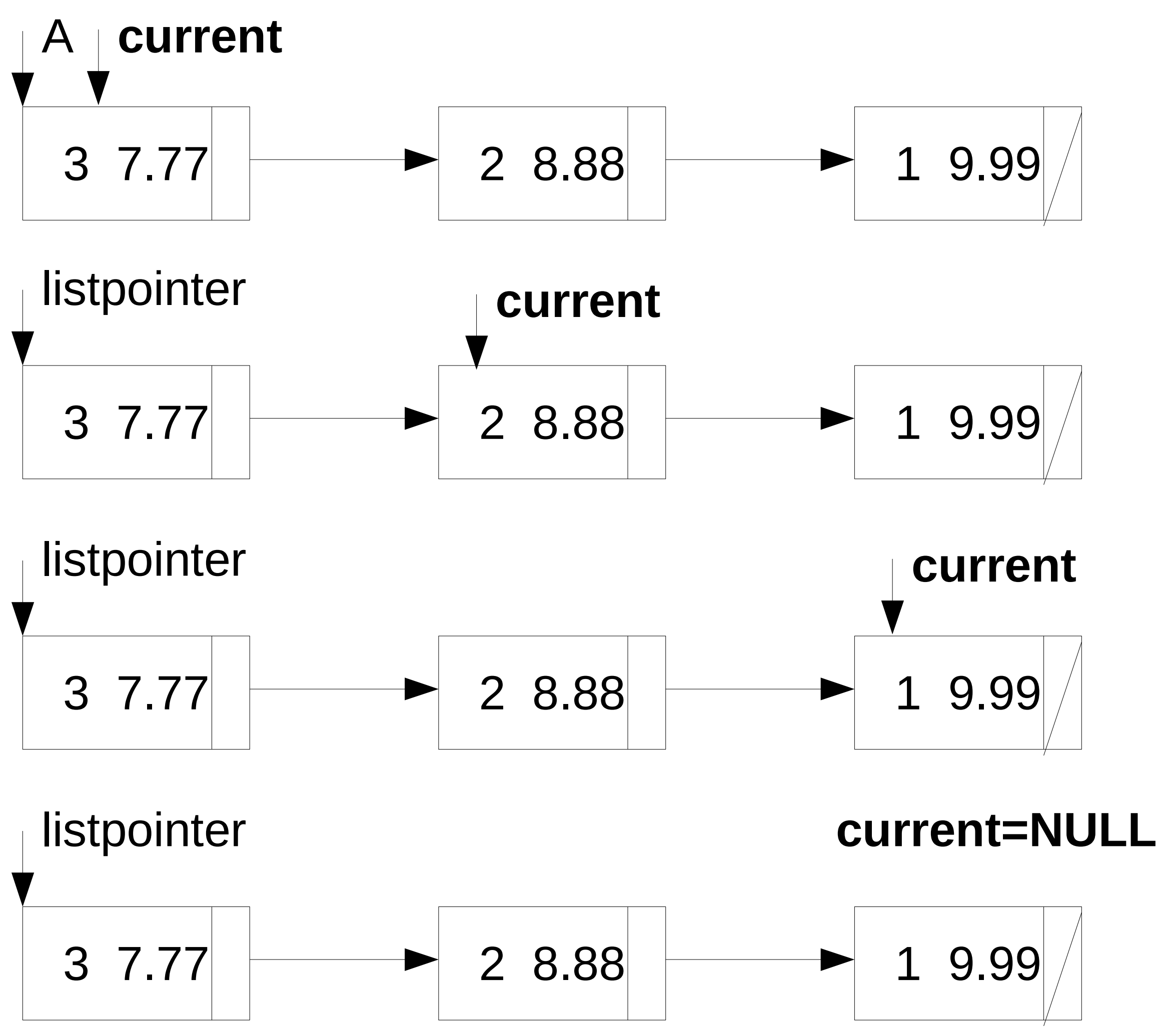
Output:

```
Account 3 balance is 7.77
Account 2 balance is 8.88
Account 1 balance is 9.99
End of the list.
```

PrintLinkedList(A) example

```
int main() {
   AddNode(A, 1, 9.99);
   AddNode(A, 2, 8.88);
   AddNode(A, 3, 7.77);
   PrintLinkedList(A);
}
```

Schematically:




```
Output:
Account 3 balance is 7.77
Account 2 balance is 8.88
Account 1 balance is 9.99
End of the list.
```

AddNode2(): insert after tail

- Create a pointer "`current`", of same type as node
- `current` initially points to the first element of the linked-list, `listpointer`
- We need to stop the `current` pointer just ***before*** the ***end*** of the list
- We can then add the new element:
- Making sure that current points to the new element
- The new element points to NULL.
- Check whether the list was empty before inserting this element!

AddNode2(): insert after tail

```
void AddNode2(Node*& listpointer, int a, float b) {
// add a new node to the TAIL of the list
  Node *current;
  current=listpointer;
  if(current!=NULL){
    while (current->next!=NULL){
      current=current->next;
    }
  }// now current points to the last element
  Node *temp;
  temp = new Node;
  temp->accnumber = a;
  temp->balance = b;
  temp->next = NULL;
  if(current!=NULL) current->next = temp;
  else listpointer=temp;
}
```

AddNode2() example

```
int main() {
  AddNode2(A, 1, 9.99);
  AddNode2(A, 2, 8.88);
  AddNode2(A, 3, 7.77);
  PrintLL(A);
}
```

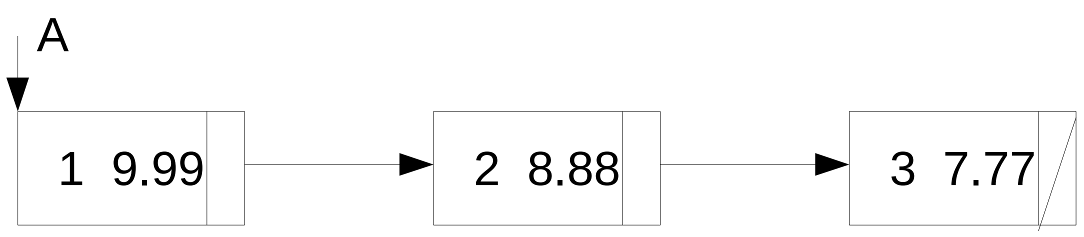
Output:

```
Account 1 balance is 9.99
Account 2 balance is 8.88
Account 3 balance is 7.77
End of the list.
```

AddNode2() example

```
int main() {
   AddNode2(A, 1, 9.99);
   AddNode2(A, 2, 8.88);
   AddNode2(A, 3, 7.77);
   PrintLL(A);
}
```

Schematically:



AddNode() X AddNode2()

AddNode() inserts elements in **reverse** order
AddNode2() inserts in the same sequence

Which one is better?

More important: which one performs better?

AddNode() inserts without traversing the LL

AddNode2() has to ***traverse*** the entire LL to find the last element...

Massey University
COLLEGE OF SCIENCES

Linked-lists Search and Remove

We know how to **add** nodes and **print** nodes of our lists.

We also need some extra functions to deal with elements, such as **Search** and **Remove**.

We need to deal with pointers appropriately to achieve that, it is easy to make a subtle mistake and crash...

Search step-by-step

- Create a pointer "`current`", of same type as node
- `current` initially points to the list, which is the first element of the linked-list
- At any point, if `current` is NULL → reached the **end** of the list (last element)
- We keep checking for `accnumber` and update `current=current->next;`
- Note that we go through the entire list, and we either find the `accnumber` we look for. Or reach the end of the list, so we print a message saying we did not find it.

Massey University
COLLEGE OF SCIENCES

Search a linked-list: pseudo-code

```
Algorithm SearchLinkedList
Required: List listpointer, integer x
Output: void
declare current (a pointer of type Node)
current = listpointer
while (true) do
   if (current == NULL) then
     break
   end if
   if (x == current->accnumber) then
      print accnumber and balance
      return
   end if
   current = current->next
end while
print error message
```

Linked-lists Search

```
void Search(Node *listpointer, int x) {
// search for the node with account number equal to x
Node *current;
  current = listpointer;
  while (true) {
    if (current == NULL) { break; }
    if (current->accnumber == x) {
      printf("Balance of %i is %1.2f\n",
        x, current->balance);
      return;
    }
    current = current->next;
  }
  printf("Account %i is not in the list.\n", x);
}
```

Search example

```
int main() {
  AddNode(A, 1, 9.99);
  AddNode(A, 2, 8.88);
  AddNode(A, 3, 7.77);
  Search(A,123);
  Search(A,1);
  Search(A,2);
  Search(A,3);
}
```

Output:

```
Account 123 is not in the list.
Balance of 1 is 9.99
Balance of 2 is 8.88
Balance of 3 is 7.77
```

Remove nodes: pseudo-code

```
Algorithm RemoveNode
Required: List listpointer, integer x
Output: void
declare current and prev (pointers to Node)
current = listpointer, prev = NULL
while (current != NULL)
  if (current->accnumber == x) then break;
  end if
  prev = current;
  current = current->next;
end while
prev->next = current->next;
delete current;
```

Note: the algorithm only works when removing nodes from lists with several nodes, where **x** exists...

Massey University
COLLEGE OF SCIENCES

Linked-lists Remove nodes

```
void Remove(Node * & listpointer, int x) {
  Node *current, *prev;//why do we need 2 pointers?
  current = listpointer;
  prev = NULL;
  while (current != NULL) {
    if (current->accnumber == x) { break; }
    prev = current;
    current = current->next;
  }
  if (prev == NULL) {
    listpointer = listpointer->next;
  } else {
    prev->next = current->next;
  }
  delete current;
}
```

RemoveNode example

```
int main() {
 AddNode(A, 1, 9.99);
 AddNode(A, 2, 8.88);
 AddNode(A, 3, 7.77);
 AddNode(A, 4, 6.66);
 AddNode(A, 5, 5.55);
 Search(A,2);
 Remove(A,2);
 Search(A,2);
}
```

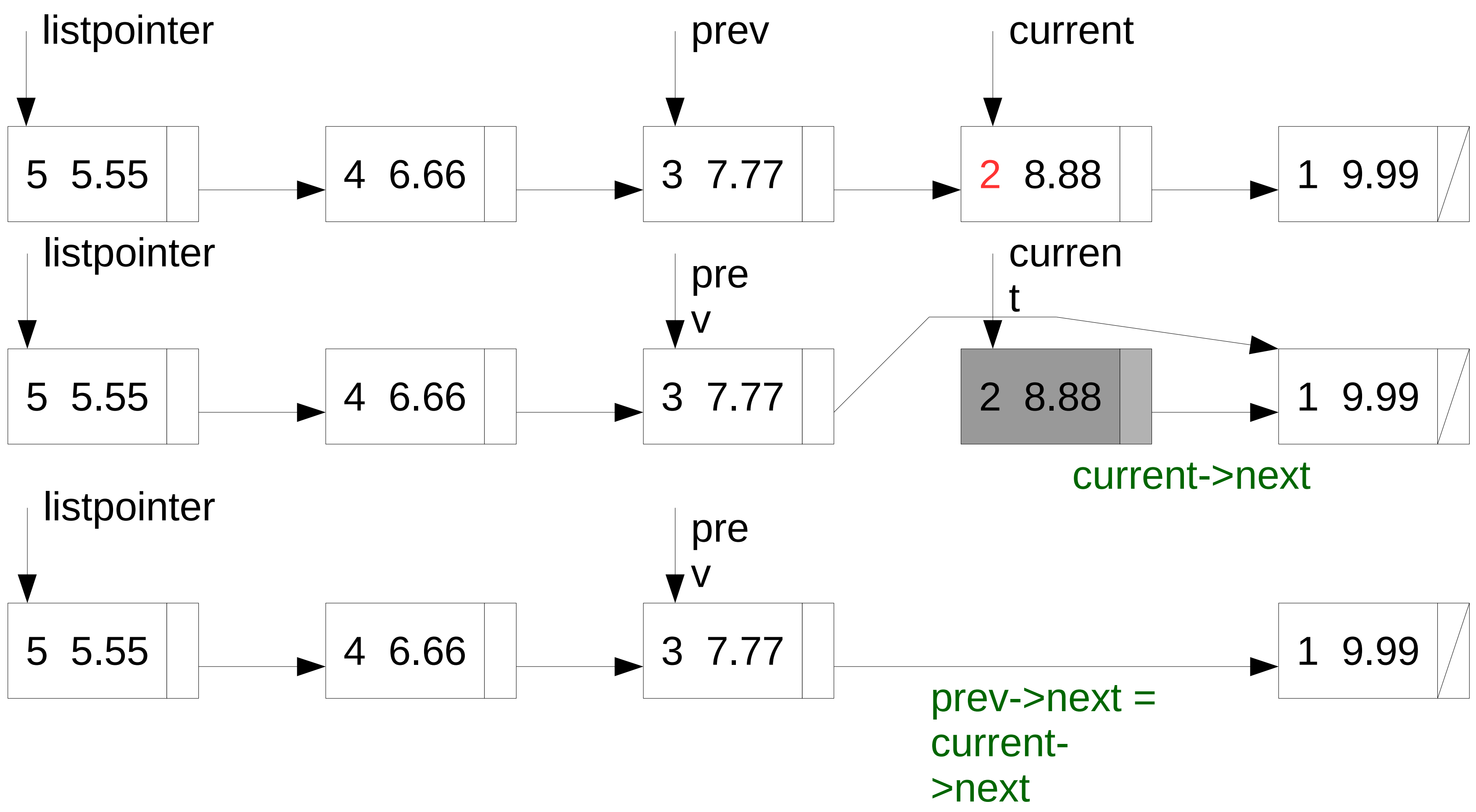
Balance of 2 is 8.88
Account 2 is not in the list.

RemoveNode step-by-step

- Two pointers, `“current”` and `“prev”`
- `current` initially points to the list
- `prev` initially points to nothing (NULL)
- While `current` is not NULL, search the list until find X. Keep swapping `prev = current`
- If X is found, change `prev` pointer to jump one element
- Now we can delete the element by deallocating `current`

RemoveNode

Schematically: removing accnumber==2:



Massey University
COLLEGE OF SCIENCES

Question: what happens if...

```
int main() {
 Node A = NULL;
 AddNode(A, 1, 9.99);
 AddNode(A, 2, 8.88);
 AddNode(A, 3, 7.77);
 AddNode(A, 4, 6.66);
 AddNode(A, 5, 5.55);
 Search(A,2);
 RemoveNode(A,2);
 Search(A,2);
 RemoveNode(A,2);//try to remove again
}
```

Balance of 2 is 8.88
Account 2 is not in the list.

Answer:

```
int main() {
 Node A = NULL;
 AddNode(A, 1, 9.99);
 AddNode(A, 2, 8.88);
 AddNode(A, 3, 7.77);
 AddNode(A, 4, 6.66);
 AddNode(A, 5, 5.55);
 Search(A,2);
 RemoveNode(A,2);
 Search(A,2);
 RemoveNode(A,2);//try to remove again
}
```

Balance of 2 is 8.88
Account 2 is not in the list.
Segmentation fault!!!

Linked-lists Remove nodes

```
void Remove(Node * & listpointer, int x) {
  Node *current, *prev;//why do we need 2 pointers?
  current = listpointer;
  prev = NULL;
  while (current != NULL) {
    if (current->accnumber == x) { break; }
    prev = current;
    current = current->next;
  }
  if (current == NULL) return;//avoid segment. fault
  if (prev == NULL) {
    listpointer = listpointer->next;
  } else {
    prev->next = current->next;
  }
  delete current;
}
```

Extra pointers

Pointers can be added to point to

rear

middle

one third etc...

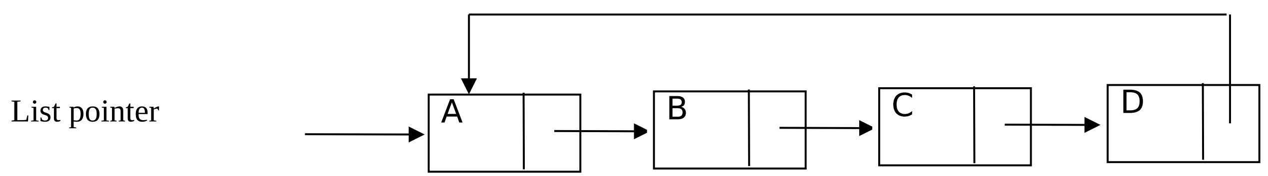
Or a combination of the above

Can any of these pointer help with the performance? How?

New search functions can be devised. What is the advantage?

More about linked-lists next week

Circular lists



Doubly-liked-lists