Massey University
COLLEGE OF SCIENCES

159201

Week 1

Summer 2014

Te Kunenga
ki Pūrehuroa

Data Types

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

Te Kunenga ki Pūrehuroa

Data Types

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

Data Types

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

Te Kunenga ki Pūrehuroa

Abstract Data Types (ADT)

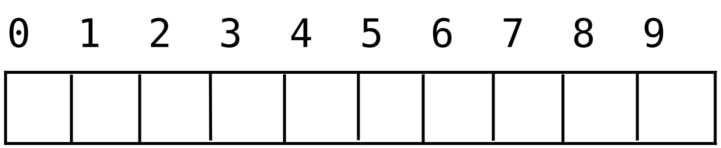
- Specification separate from implementation
- Example:
  - A book record consists of:
    - Title (max 40 characters)
    - Call number (real)

Abstract Data Types (ADT)

- Advantages of ADTs
  - Reduce details – allow focus to be on the "main picture"
  - Different implementations can be used – e.g., array or linked-list
  - Underlying implementation can be changed or upgraded
  - 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]



These are the **index** numbers

Revision of Arrays

Advantages of Arrays

Simple, Fast, Random access

Disadvantages of Arrays

Every element if of the same data type

Fixed size – too small or too big at runtime

Difficult to insert or delete without leaving spaces

2D arrays

Example:

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

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

	0	1	2	3
0				
1				
2			64	
3				

Massey University
COLLEGE OF SCIENCES

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

Te Kunenga ki Pūrehuroa

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("%ld ",&matrix[a][b]);//pointers

                }

                printf("\n");

        }

}
```

Te Kunenga ki Pūrehuroa

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

Reference and pointers

Remember:

*	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 (that points to a structure)

Examples with *

```c
#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 *&)

```
#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: ? ? ?

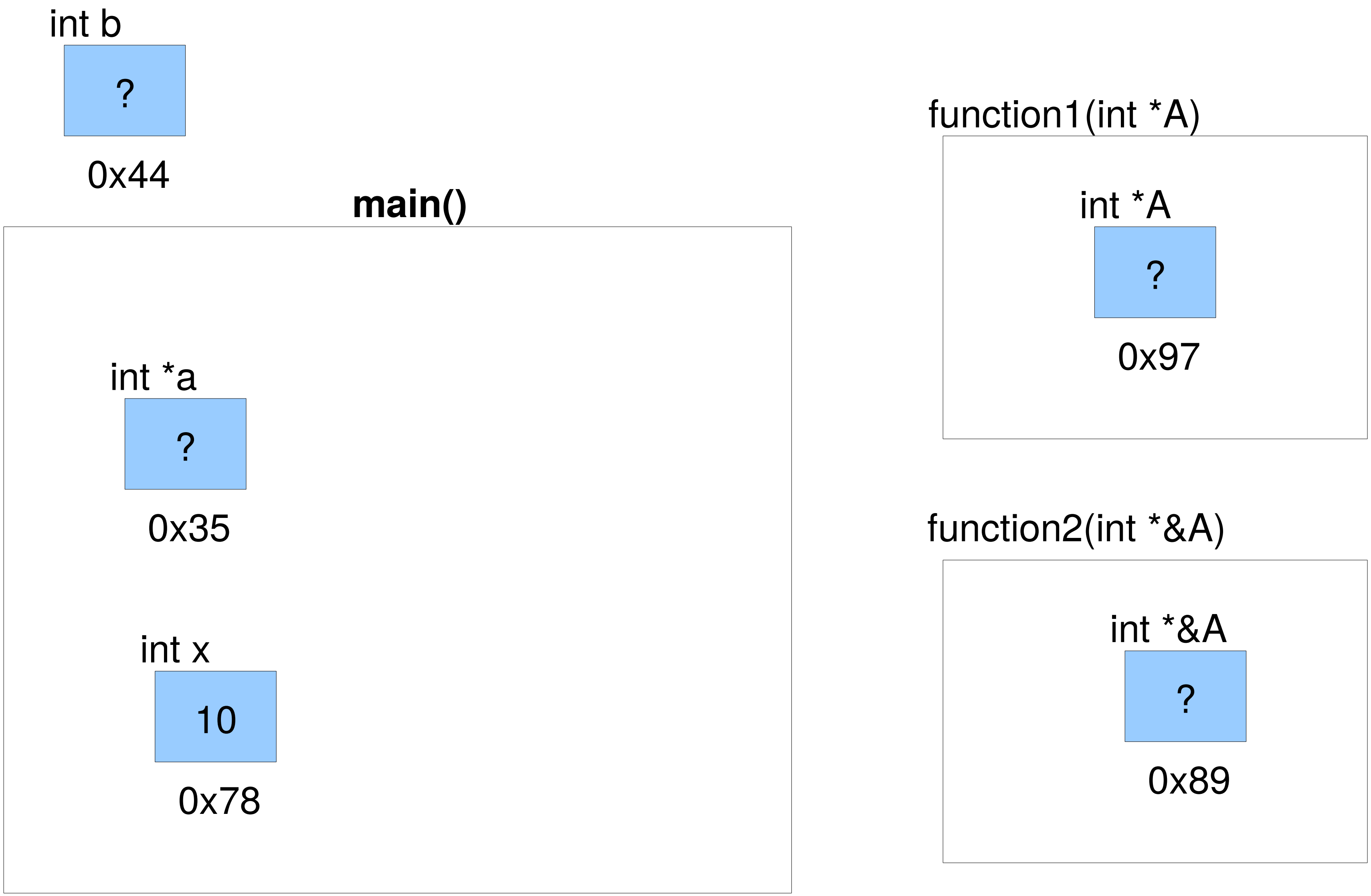
Examples with funct(type *&)

```
#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

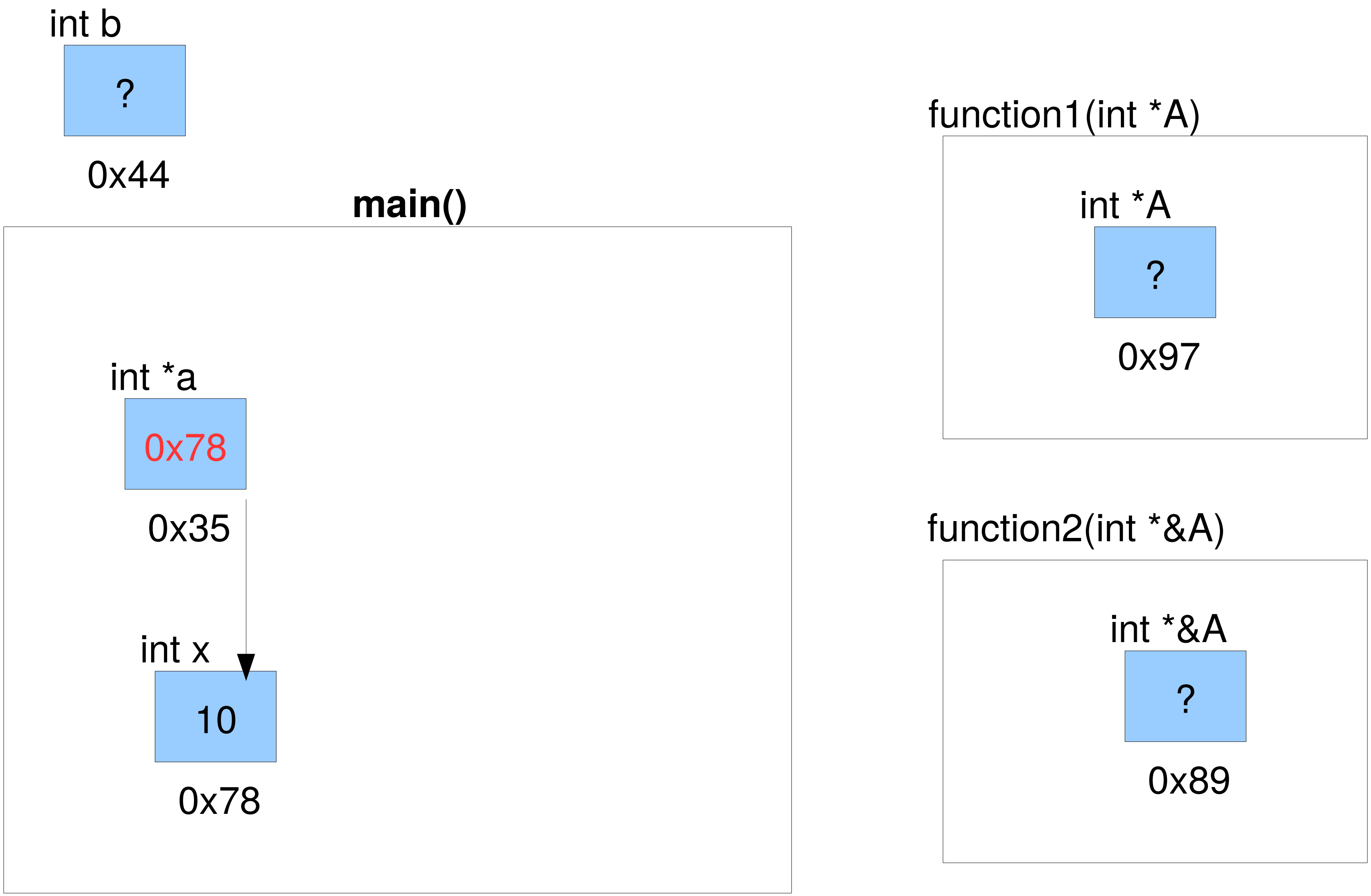
NOT: 10 20 30

State of memory at line 7



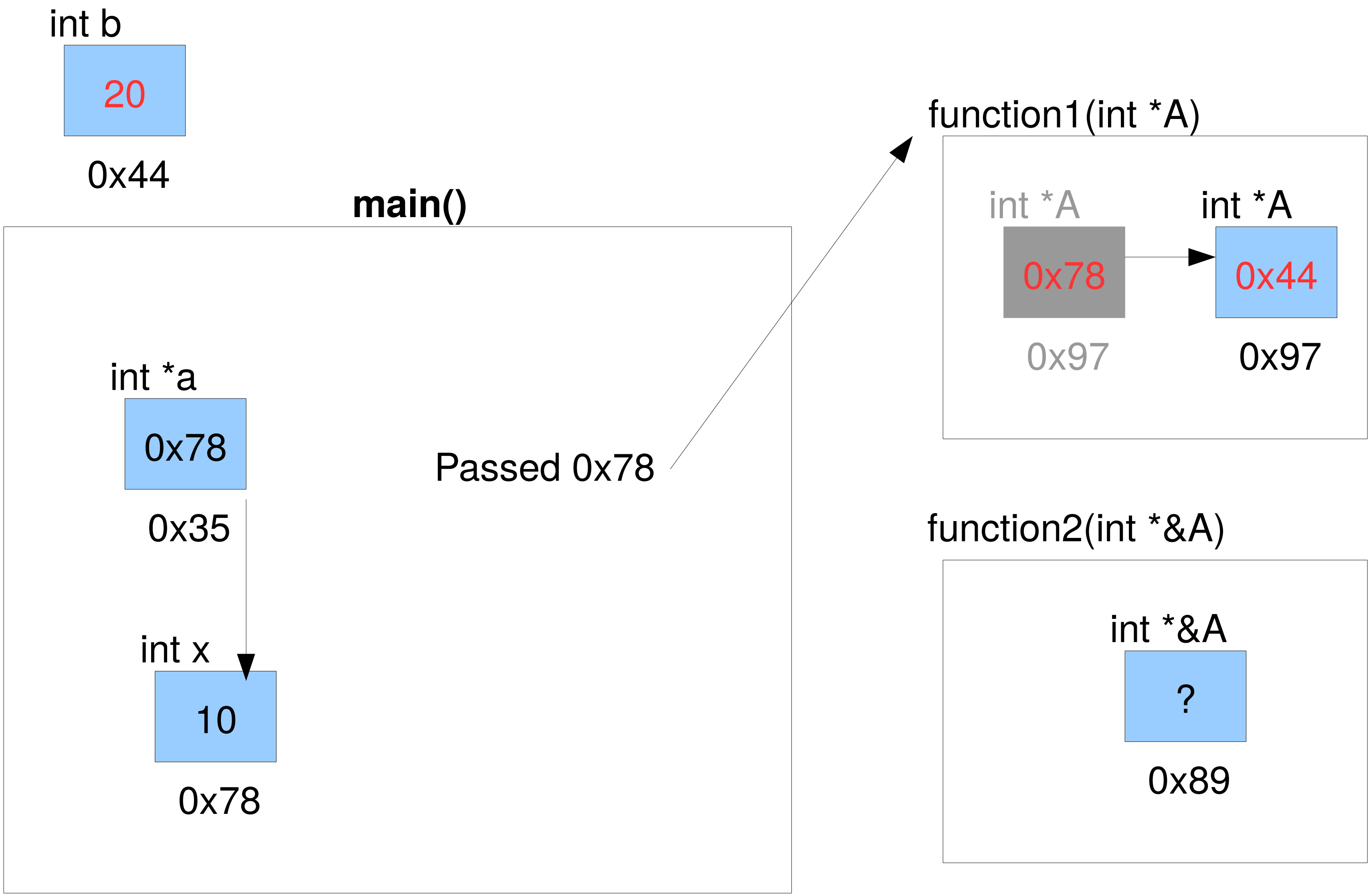
Massey University
COLLEGE OF SCIENCES

State of memory at line 9



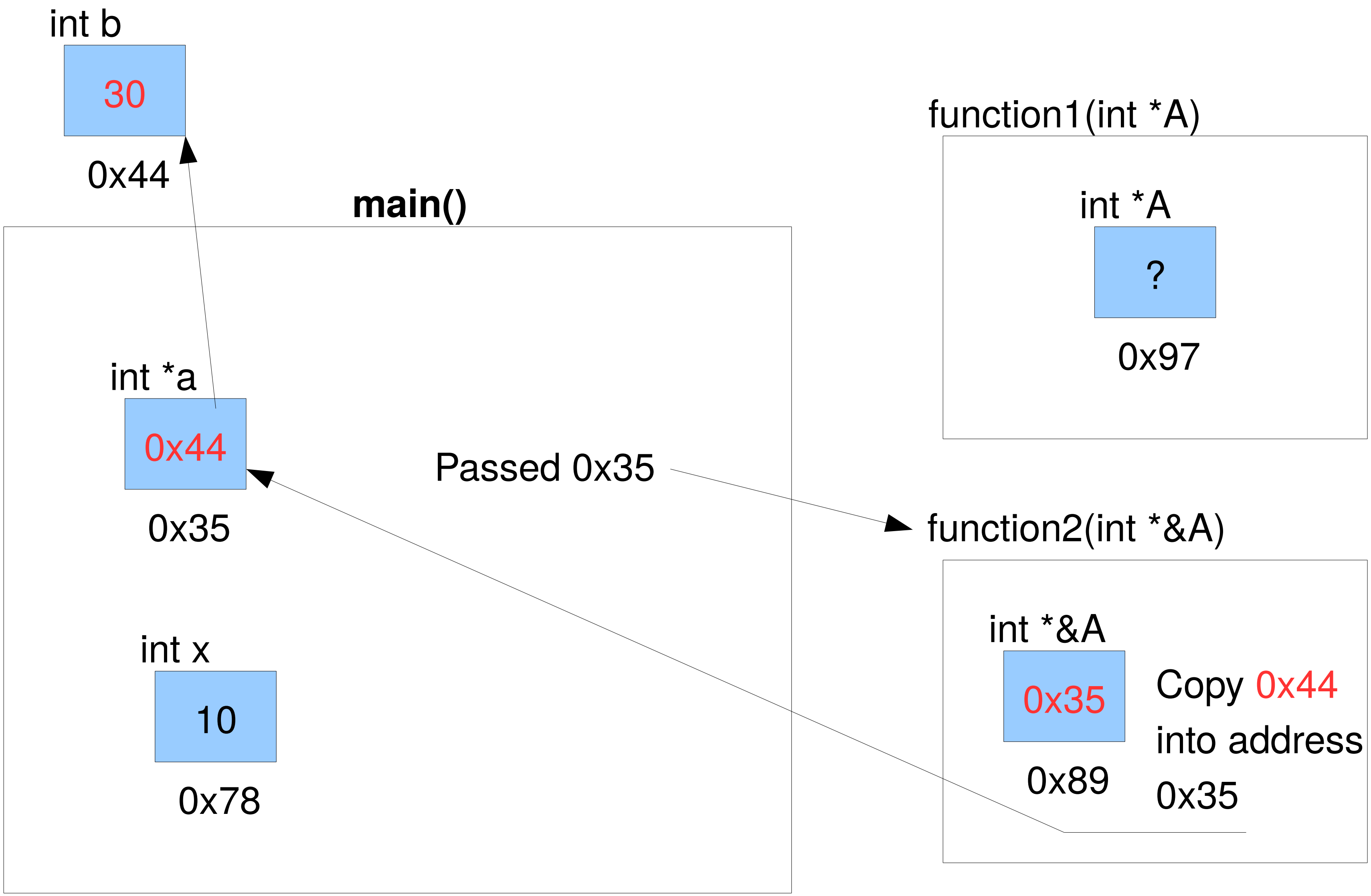
Te Kunenga ki Pūrehuroa

State of memory at line 12



Massey University
COLLEGE OF SCIENCES

State of memory at line 15



Massey University
COLLEGE OF SCIENCES

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.

Te Kunenga ki Pūrehuroa

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;
}
```

Te Kunenga ki Pūrehuroa

L02

Linked-lists

Linked-lists are sequences of connected nodes.

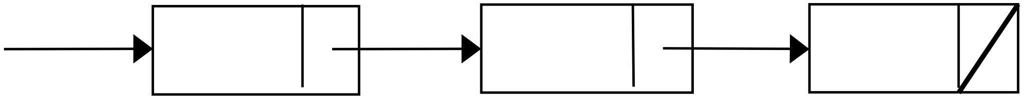
Linked-lists are empty at the start.

Nodes are added dynamically (at runtime).

Nodes contain pointers to other nodes.

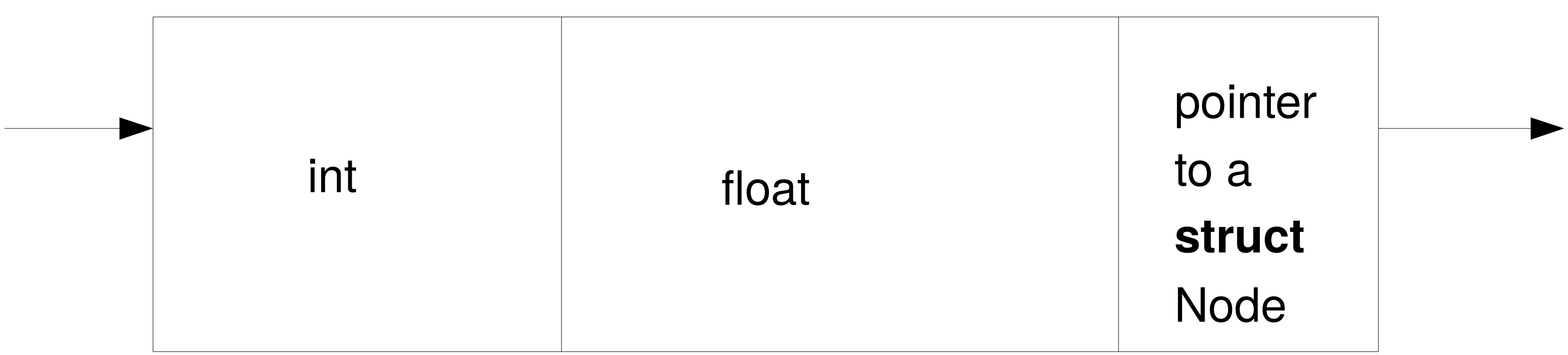
The address of the list is the pointer to the first node.

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



Linked-lists

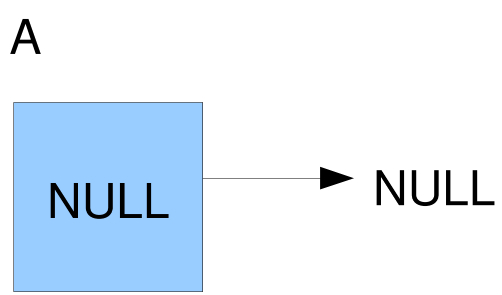
```
struct Node {  //declaration
  int accnumber;
  float balance;
  struct Node *next;
};
typedef struct Node Node;
//this should reserve memory space for this struct...
```



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

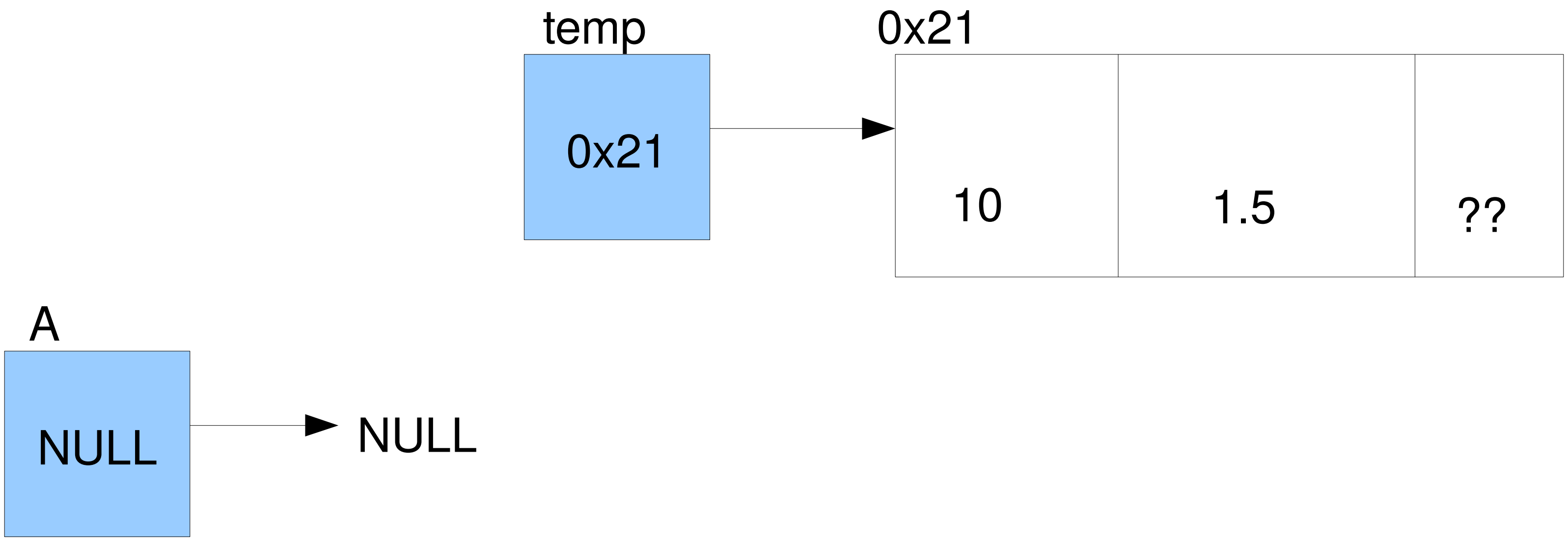
Lets add, manually, a new node on list **A:**

Node *temp; //declare a temporary pointer to a Node

temp = **new** Node;//allocate space

temp->accnumber=10;//load the values

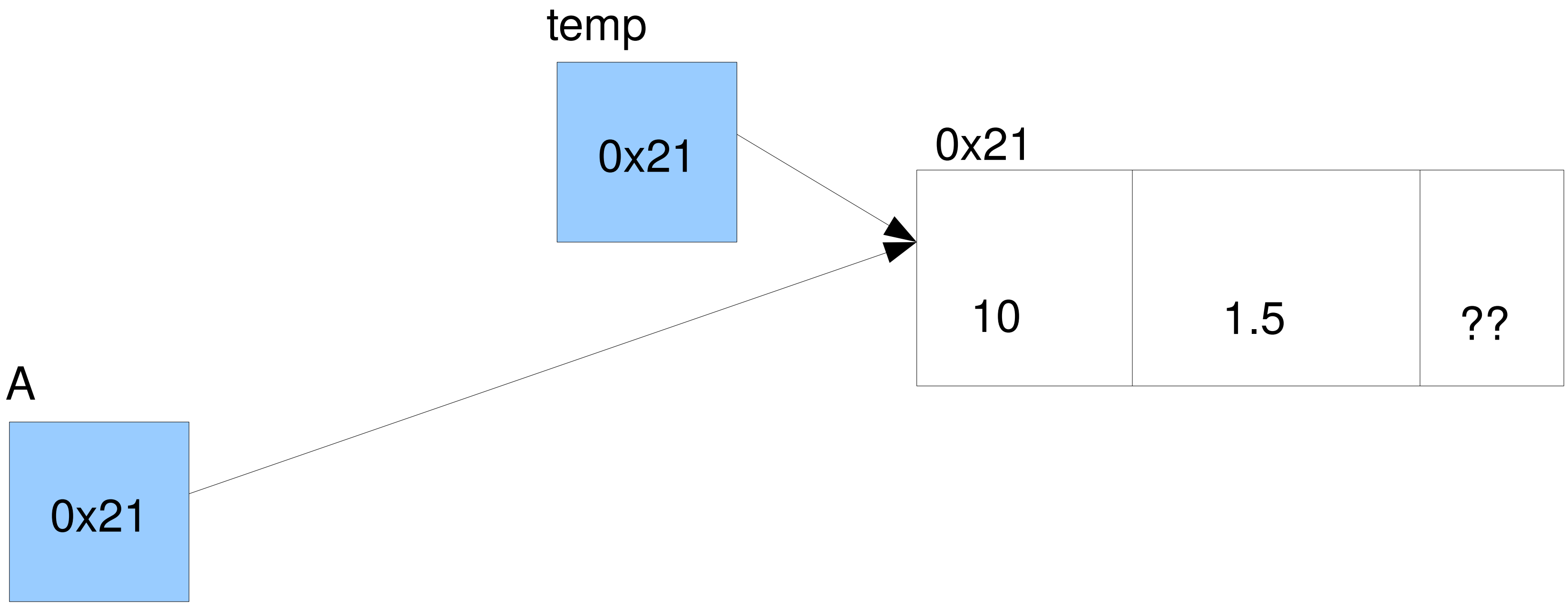
temp->balance=1.5;



Linked-lists

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

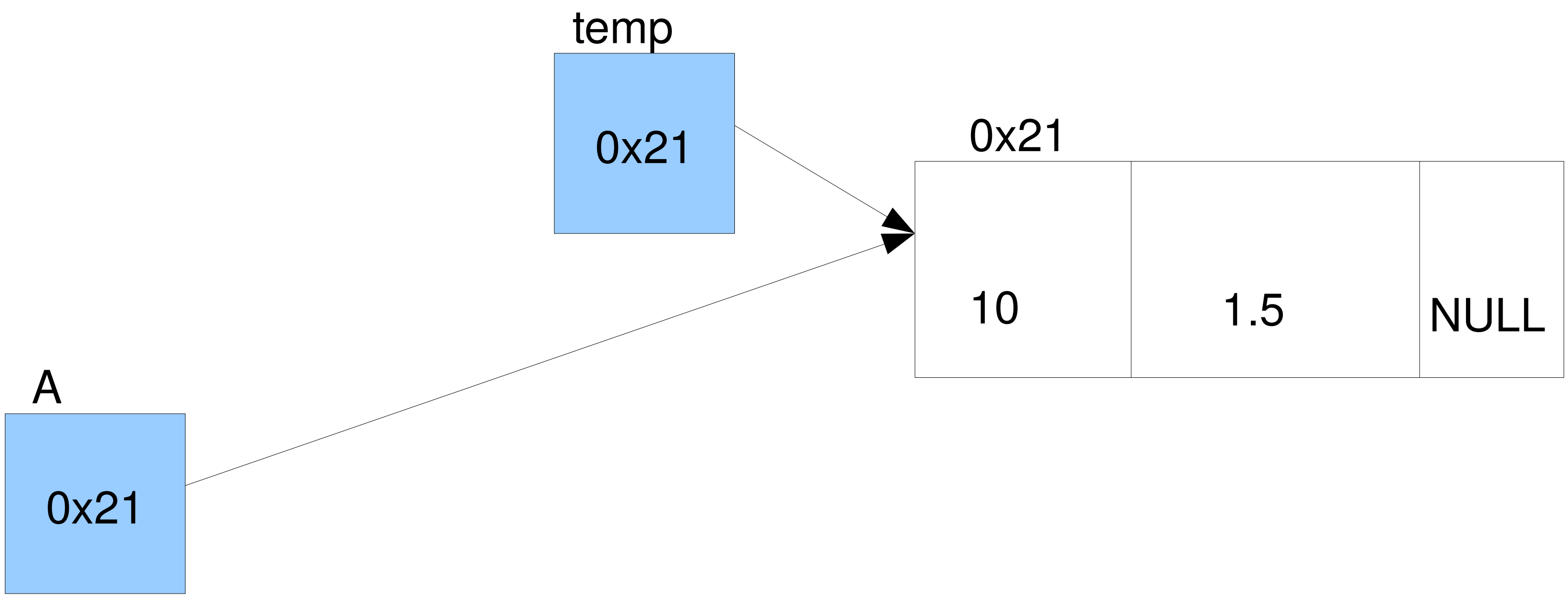
```
A = temp;
```



Linked-lists

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

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



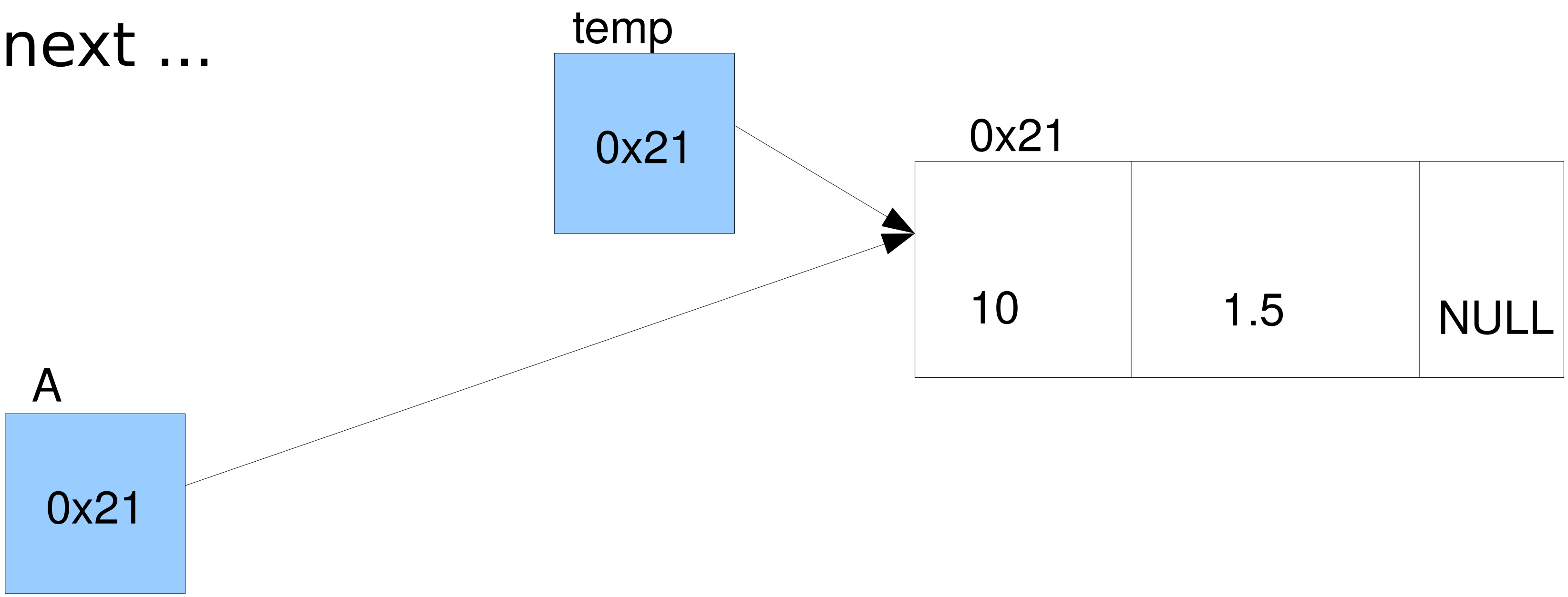
Linked-lists

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

A->accnumber

A->balance

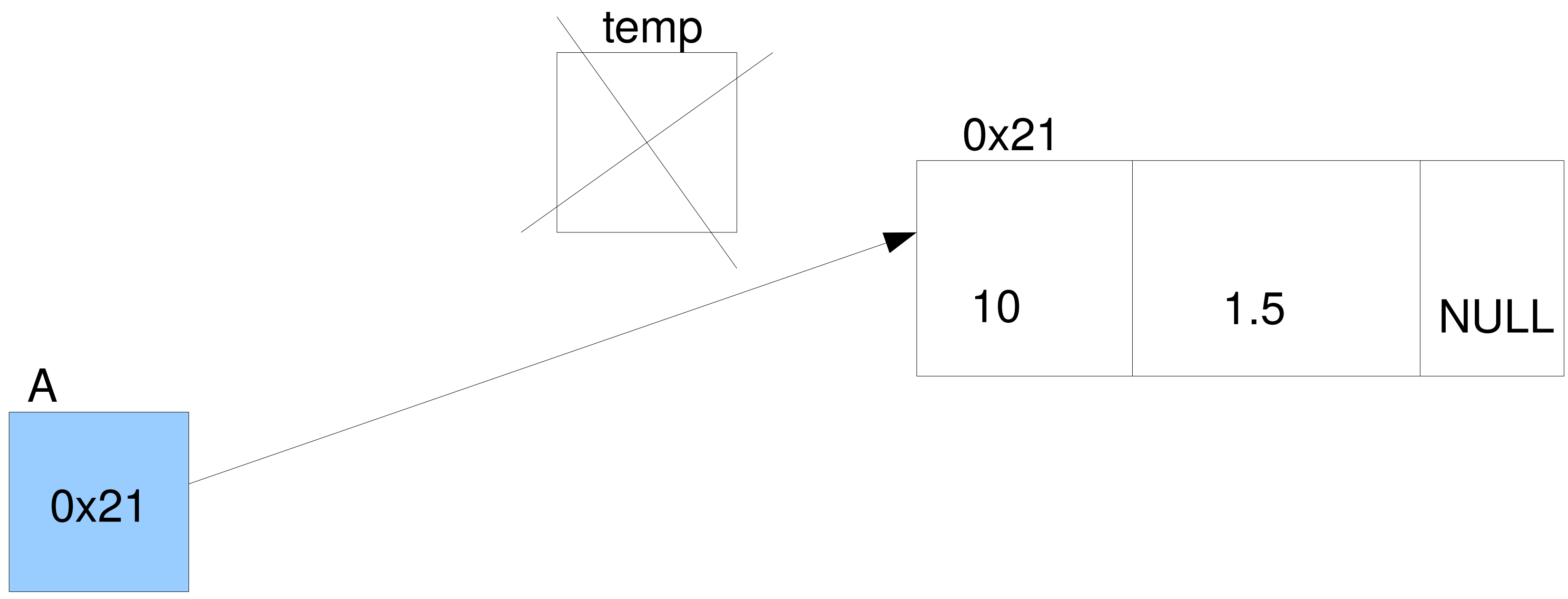
A->next ...



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...

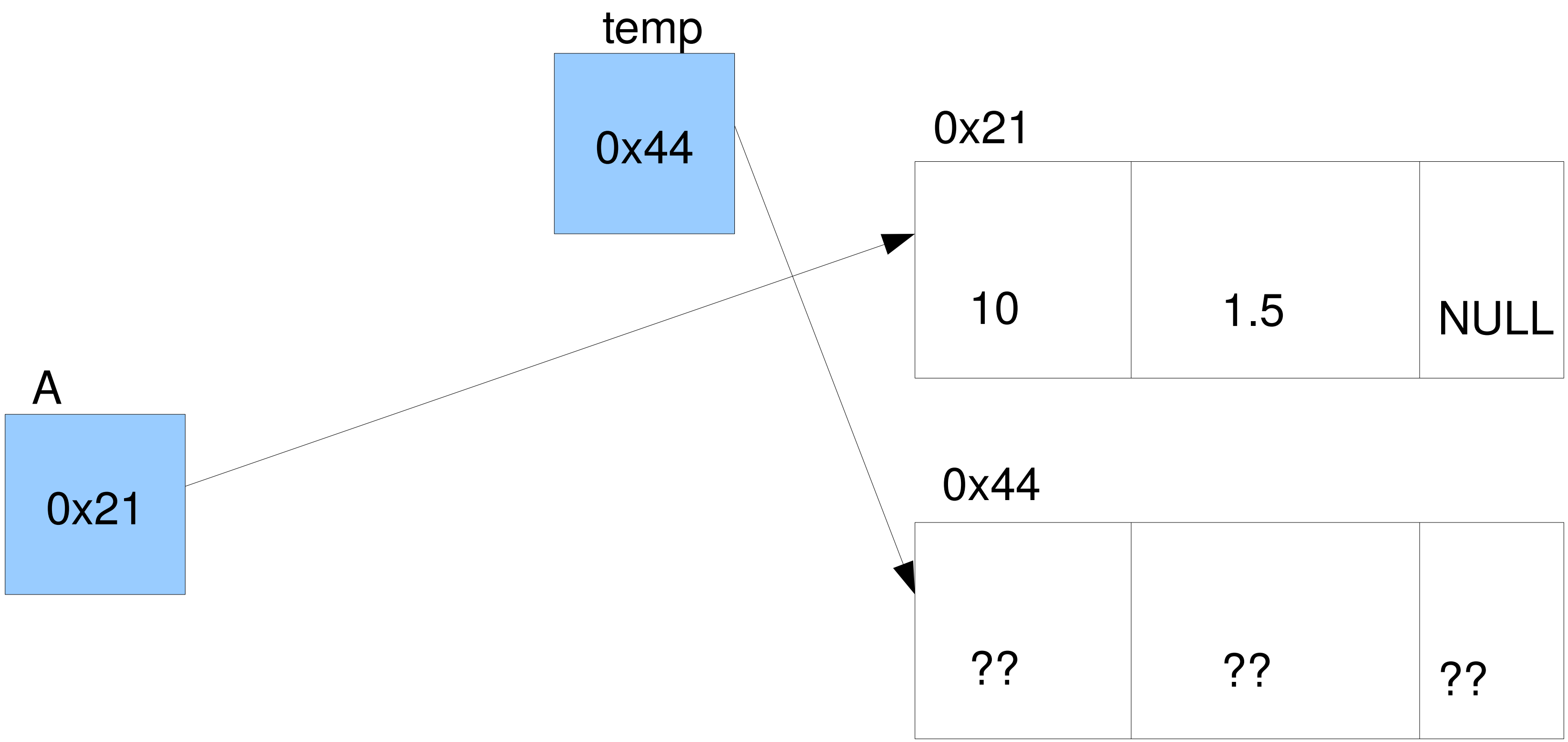


Te Kunenga ki Pūrehuroa

Linked-lists

Suppose you want a second element linked to list A.

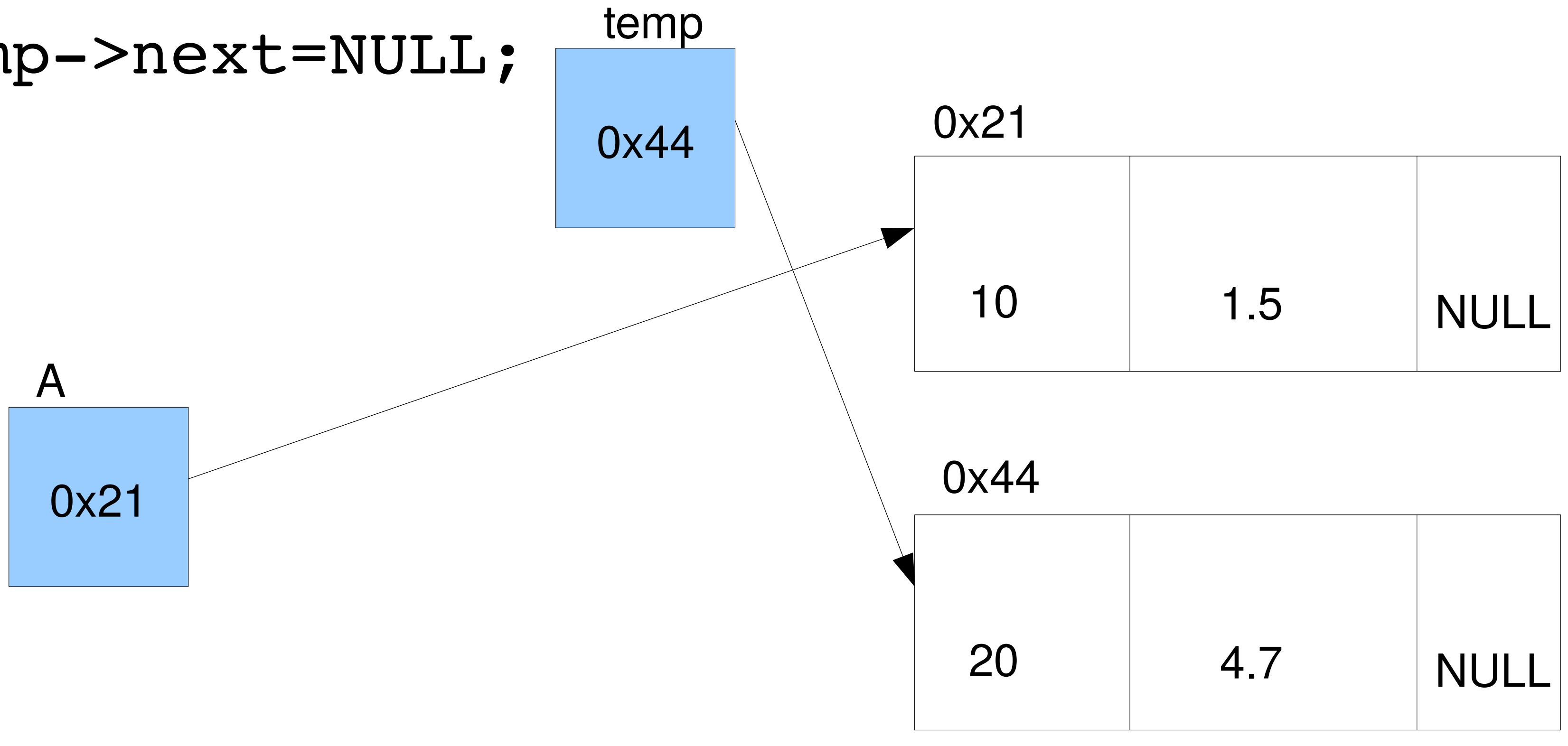
```
temp = new Node;
```



Linked-lists

Load values to it:

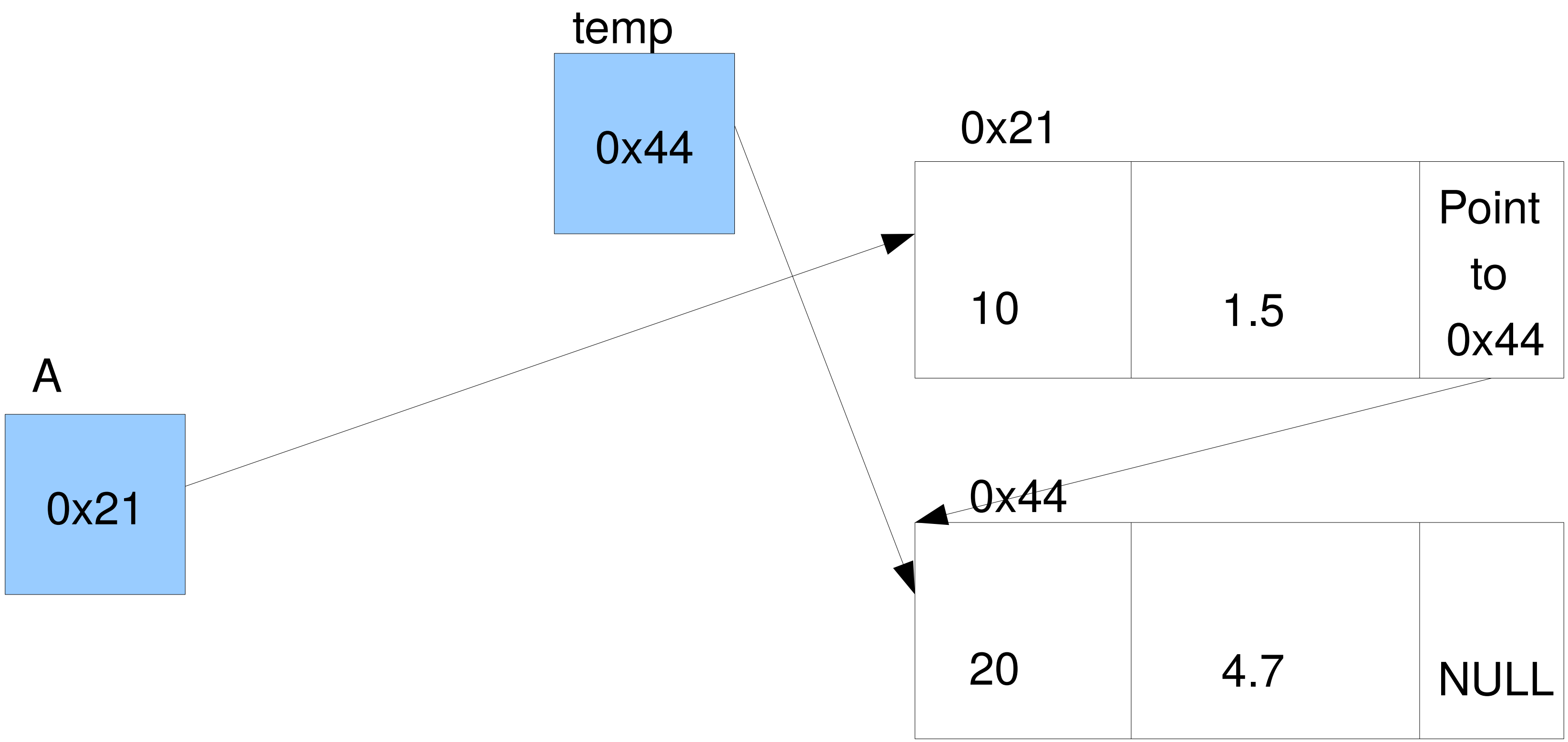
```
temp->accnumber=20;//load the values
temp->balance=4.7;
temp->next=NULL;
```



Linked-lists

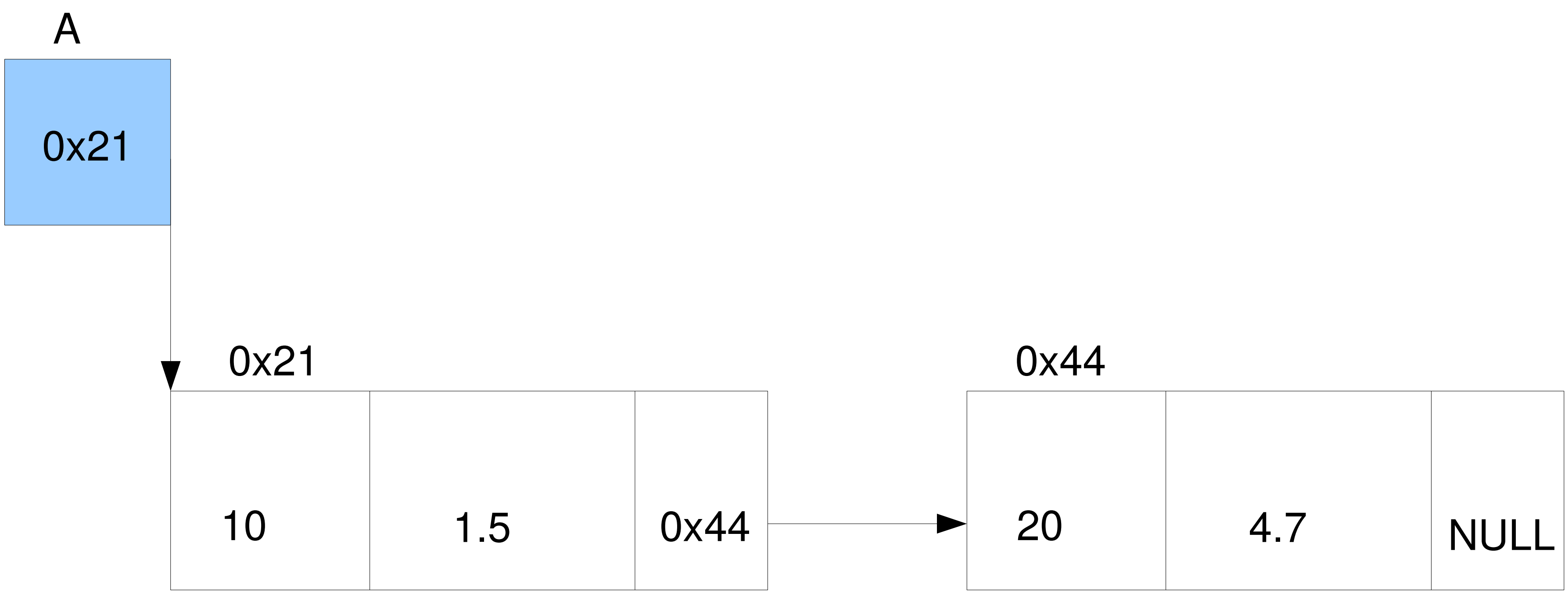
Then, link the second element to the first:

```
A->next=temp;
```



Linked-lists

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



Discussion: what happens if we create more elements?

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

Sample Linked-list in C++

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

void AddNode(Node * & listpointer, int a, float b);

int main() {
   A = NULL;  // ALL linked-lists start empty
   B = NULL;
   AddNode(A, 123, 99.87);
   AddNode(B, 789, 52.64);
}
```

AddNode()

```
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;
}
```

But wait a minute…

This will not produce the same linked-list as before, as it will be **inverted** if we add in the same order!

Can you think of two ways of adding nodes, at the HEAD or at the TAIL of the linked list?

Reference and pointers

Subtle syntax in C/C++ can cause errors

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...*

Te Kunenga ki Pūrehuroa

Challenges:

1) Modify the AddNode() function (add to HEAD) to add to the TAIL of the linked-list.

2) Modify the AddNode() function to add an element AFTER a certain element (by value or position of the element).

You will need to find the last element of the linked-list by modifying the Search() function.

The answers are on Stream, study these solutions carefully and understand exactly how to control Nodes: *add, delete and search for any Node*.

Massey University
COLLEGE OF SCIENCES

L03

Te Kunenga ki Pūrehuroa

Linked-lists Search and Remove

We know how to add nodes to our lists, but just adding to the HEAD. How do we find the TAIL of a linked list?

We also need some extra functions to deal with elements, such as Search. Also, a function to delete or remove nodes that we no longer need.

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

Linked-lists Search

Search function:

```
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);
}
```

Linked-lists Search

Search function:

```
int main() {
   A = NULL;   // ALL linked-lists start empty
   AddNode(A, 1, 9.87);
   AddNode(A, 2, 8.87);
   AddNode(A, 3, 7.87);
   Search(A,123);
   Search(A,1);
   Search(A,2);
   Search(A,3);
}
```

Account 123 is not in the list.

Balance of 1 is 9.87

Balance of 2 is 8.87

Balance of 3 is 7.87

Massey University
COLLEGE OF SCIENCES

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

Te Kunenga ki Pūrehuroa

Linked-lists Remove nodes

RemoveNode function:

```
void RemoveNode(Node * & listpointer, int x) {
// remove the node containing account number x
Node *current, *prev;
  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;
}
```

Te Kunenga ki Pūrehuroa

Massey University
COLLEGE OF SCIENCES

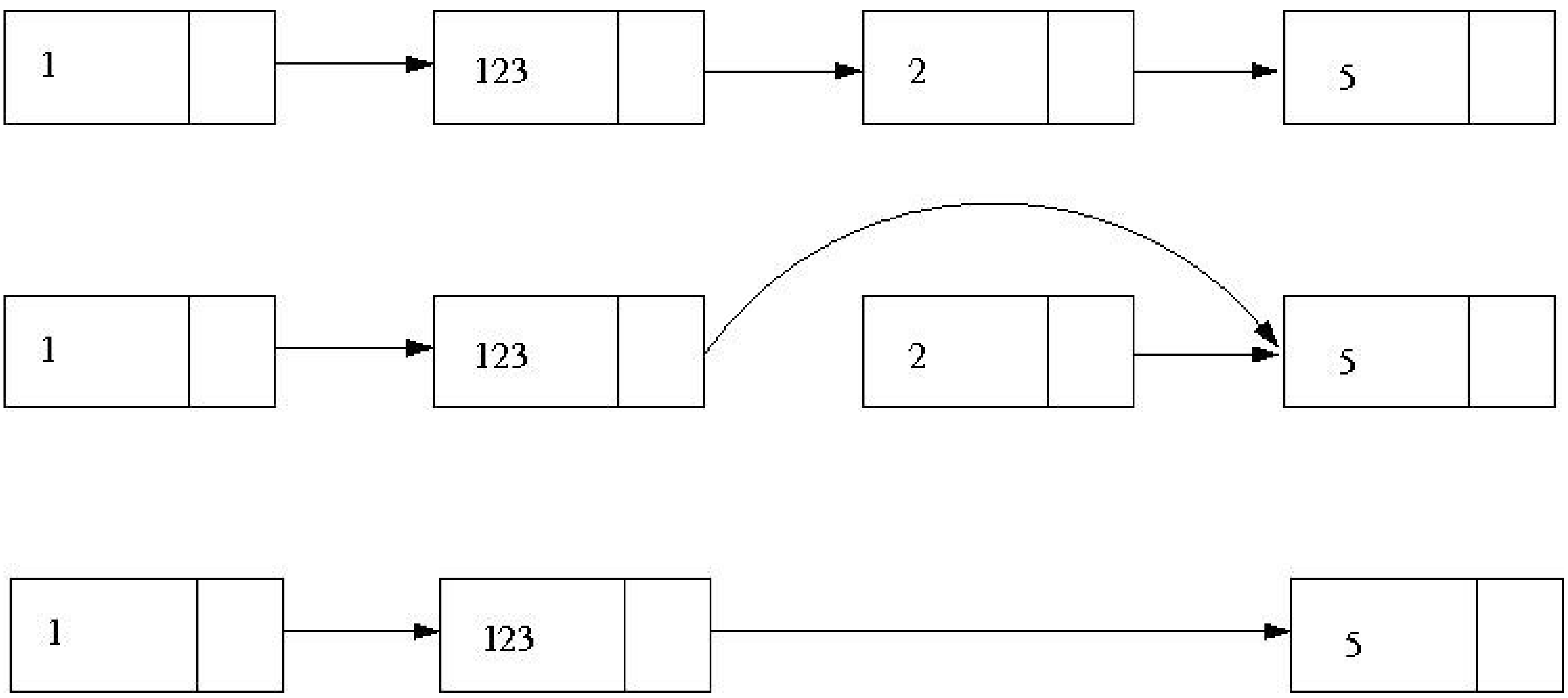
RemoveNode step-by-step

- 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`

Te Kunenga ki Pūrehuroa

RemoveNode in pictures

Suppose we want to remove element accnumber==2:



RemoveNode example

Usage

```
int main() {
  A = NULL;  // ALL linked-lists start empty
  AddNode(A, 1, 9.87);
  AddNode(A, 123, 8.87);
  AddNode(A, 2, 7.87);
  AddNode(A, 5, 7.87);
  Search(A,2);
  RemoveNode(A,2);
  Search(A,2);
}
```

Balance of 2 is 7.87

Account 2 is not in the list.

Question!

What happens if:

```
int main() {
  A = NULL;  // ALL linked-lists start empty
  AddNode(A, 1, 9.87);
  AddNode(A, 123, 8.87);
  AddNode(A, 2, 7.87);
  AddNode(A, 5, 7.87);
  Search(A,2);
  RemoveNode(A,2);
  Search(A,2);
  RemoveNode(A,2);
}
```

Te Kunenga ki Pūrehuroa

Question!

What happens if:

```
int main() {
  A = NULL;  // ALL linked-lists start empty
  AddNode(A, 1, 9.87);
  AddNode(A, 123, 8.87);
  AddNode(A, 2, 7.87);
  AddNode(A, 5, 7.87);
  Search(A,2);
  RemoveNode(A,2);
  Search(A,2);
  RemoveNode(A,2);
}
```

Balance of 2 is 7.87

Account 2 is not in the list.

Segmentation fault!!!!

Extra pointers

Pointers can be added to point to

- rear
- middle
- one third etc...

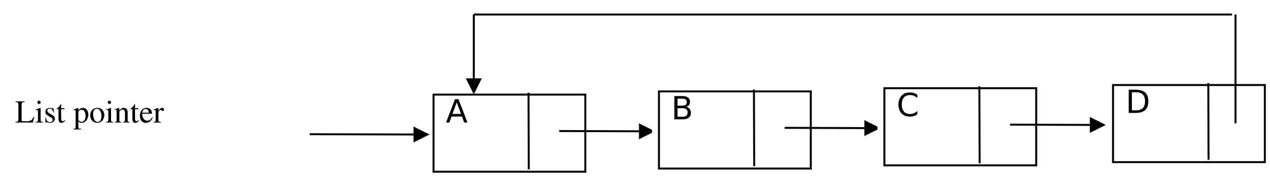
Or a combination of the above

Extra operations on the AddNode() and RemoveNode()

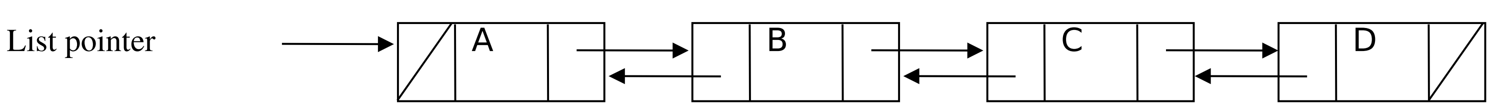
New search functions can be devised. What is the advantage?

Other types of linked-lists

Circular lists



Doubly-liked-lists



Massey University
COLLEGE OF SCIENCES

L04

Te Kunenga ki Pūrehuroa

Print all elements of a LL

Simple approach: scan LL until the end

```
void PrintLL(Node *listpointer) {
// print all elements
Node *current;
  current = listpointer;
  int element=1;
  while (true) {
    if (current == NULL) { break; }
    printf("Element %d: Balance of acc %i is %1.2f\n",
           element, current->accnumber, current->balance);
    current = current->next;
    element++;
  }
  printf("End of the list.\n");
}
```

Te Kunenga ki Pūrehuroa

More operations with Linked-lists

Extra Operations:

Concatenate → join two separate lists

Reverse → invert the order of the elements

Split → separate the list in two

Insert a new node **after** a certain element

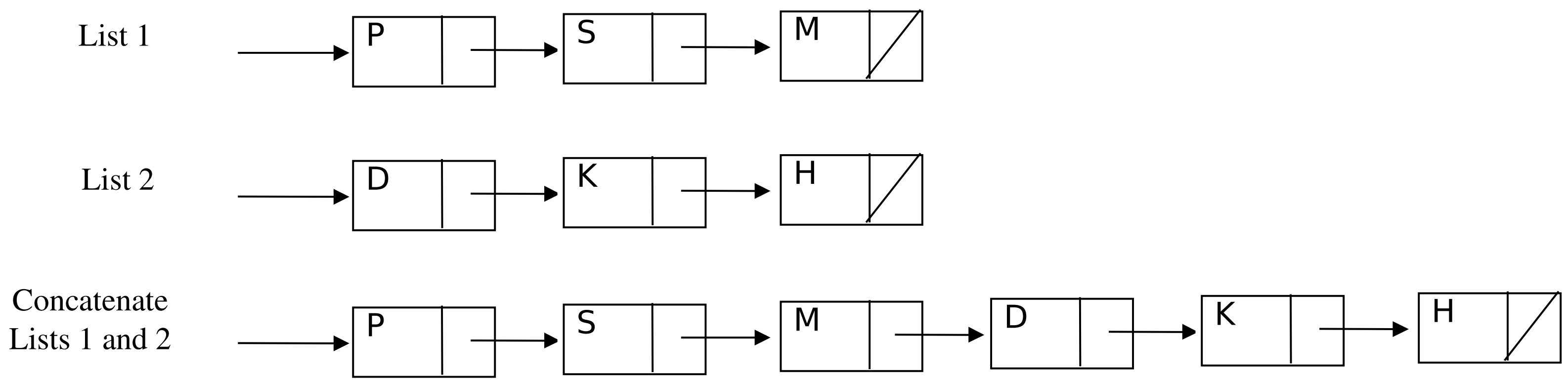
Delete by element **order** (say, the 5th element) rather than by a known key

Te Kunenga ki Pūrehuroa

Concatenate

Concatenate (join two separate lists)

The final pointer of list 1 should now point to list 2 first element



Concatenate example

- Scan listpointer1 to find the last element
- Join

```
void Concatenate(Node * &listpointer1, Node * listpointer2) {
	//find the last element of listpointer1, then join with listpointer2
	Node *current, *prev;
	current = listpointer1;
	prev = NULL;
	while (current != NULL) {
		prev = current;
		current = current->next;
	}
	if (prev == NULL) {
		//in this case listpointer1 is empty
		printf("list1 was empty, join anyway\n");
		listpointer1 = listpointer2;
	} else {
		//join lists
		printf("join\n");
		prev->next=listpointer2;
	}
}
```

Concatenate example

Main

```
int main() {
        AddNode(A, 1, 9.87);
        AddNode(A, 2, 8.87);
        AddNode(A, 3, 7.87);
        AddNode(B, 4, 6.97);
        AddNode(B, 5, 3.33);
        AddNode(B, 6, 5.78);
        PrintLL(A);
        PrintLL(B);
        Concatenate(A,B);
        PrintLL(A);
}
```

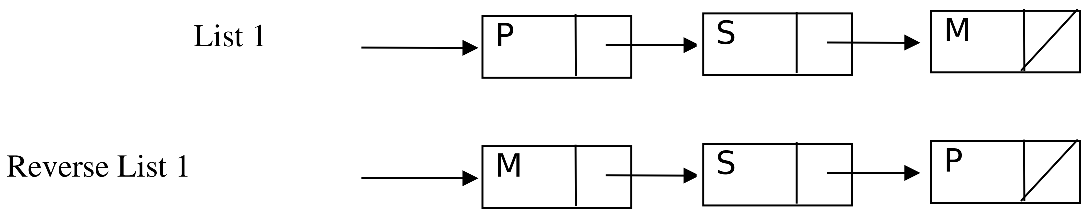
Results

Element 1: Balance of acc 3 is 7.87
Element 2: Balance of acc 2 is 8.87
Element 3: Balance of acc 1 is 9.87
End of the list.
Element 1: Balance of acc 6 is 5.78
Element 2: Balance of acc 5 is 3.33
Element 3: Balance of acc 4 is 6.97
End of the list.
join
Element 1: Balance of acc 3 is 7.87
Element 2: Balance of acc 2 is 8.87
Element 3: Balance of acc 1 is 9.87
Element 4: Balance of acc 6 is 5.78
Element 5: Balance of acc 5 is 3.33
Element 6: Balance of acc 4 is 6.97
End of the list.

Reverse

Invert the order of all the elements.

The pointer to the last element becomes the list, the the pointer to the list becomes the last element...



Reverse

Many ways of achieving that...

E.g., two methods

Method 1: create a new LL, scan once to find how many elements, copy the last one, and keep adding nodes and scanning again. Copy address and delete the original.

Method 2: Scan once, swap the contents of the last element with the first one, keep going until swap the middle elements.

Reverse

We need code to search by position:

```
Node * SearchByPosition(Node *listpointer, int x) {
  Node *current;
  current = listpointer;
  int pos=0;
  while (true) {
    if (current == NULL) { break; }
    if (pos == x) {     return current;    }
    current = current->next;
    pos++;
  }
  printf("There are only %d elements in this list\n", x);  return NULL;
}
```

Te Kunenga ki Pūrehuroa

Reverse method 1

```
void ReverseLL1(Node * &listpointer) {
  Node *current;
  Node *prev;
  Node *reversedcopy=NULL;
  current = listpointer;
  int numbelements=0;
  while (true) {//scan once
    if (current == NULL) { break; }
    prev = current;
    current = current->next;
    numbelements++;
  }
  if(numbelements!=0){
    for(int count=0;count<numbelements;count++){
        Node *temp=SearchByPosition(listpointer, count);//find contents
        if(temp!=NULL) AddNode(reversedcopy,temp->accnumber,temp->balance);//copy contents
    }
    listpointer=reversedcopy;
    return;
  }
  else {
    printf("the list is empty, nothing to reverse\n");
    return;
  }

}
```

Te Kunenga
ki Pūrehuroa

Reverse method 1

```
int main() {
  A = NULL;
  AddNode(A, 1, 9.87);
  AddNode(A, 2, 8.87);
  AddNode(A, 3, 7.87);
  AddNode(A, 4, 6.97);
  PrintLL(A);
  ReverseLL1(A);
  PrintLL(A);
}
```

Element 1: Balance of acc 4 is 6.97
Element 2: Balance of acc 3 is 7.87
Element 3: Balance of acc 2 is 8.87
Element 4: Balance of acc 1 is 9.87
End of the list.
the list is reversed
Element 1: Balance of acc 1 is 9.87
Element 2: Balance of acc 2 is 8.87
Element 3: Balance of acc 3 is 7.87
Element 4: Balance of acc 4 is 6.97
End of the list

Reverse method 1

Question 1: what is missing in ReverseLL1?

This is known as a ***memory leaking*** problem…

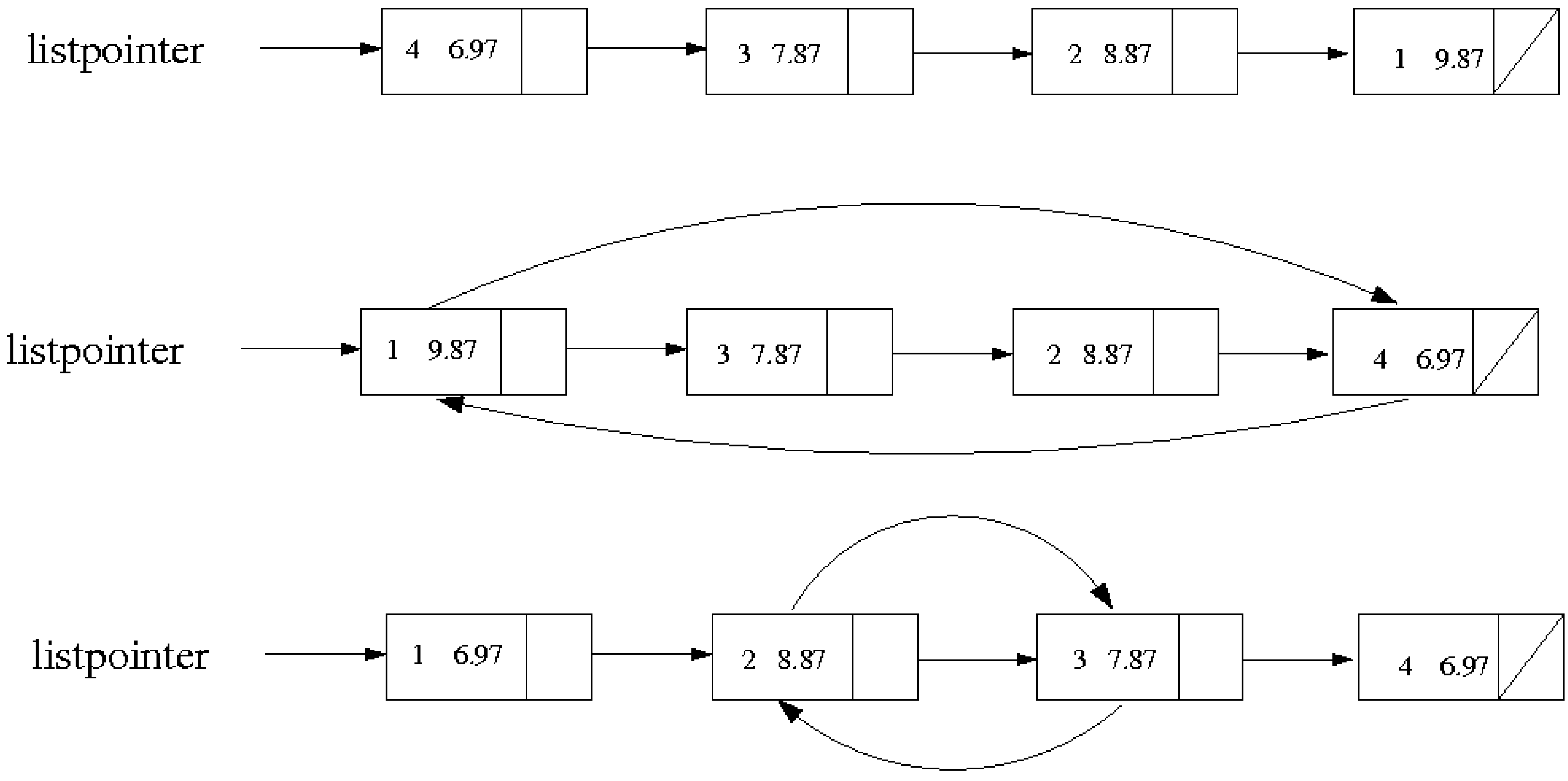
Question 2: is that efficient? How many scans/access do we have to do?

Reverse method 2

```
void ReverseLL2(Node * listpointer) {//Note we don't need & here
  Node *current, *prev, *temp, *temp2;
  current = listpointer;
  int numbelements=0;
  while (true) {//scan once
    if (current == NULL) { break; }
    prev = current;
    current = current->next;
    numbelements++;
  }
  if(numbelements!=0){
    for(int count=0;count<numbelements/2;count++){
        temp=SearchByPosition(listpointer, count);
        temp2=SearchByPosition(listpointer, numbelements-1-count);
        //swap values
        int accnumber_temp=temp->accnumber;
        float balance_temp=temp->balance;
        temp->accnumber = temp2->accnumber;
        temp->balance = temp2->balance;
        temp2->accnumber = accnumber_temp;
        temp2->balance = balance_temp;
    }
    printf("the list is reversed\n");
    return;
  }
  else { printf("the list is empty, nothing to reverse\n"); return;  }
}
```

Te Kunenga ki Pūrehuroa

Reverse method 2



Circular Linked-lists

The last element points back to the first one

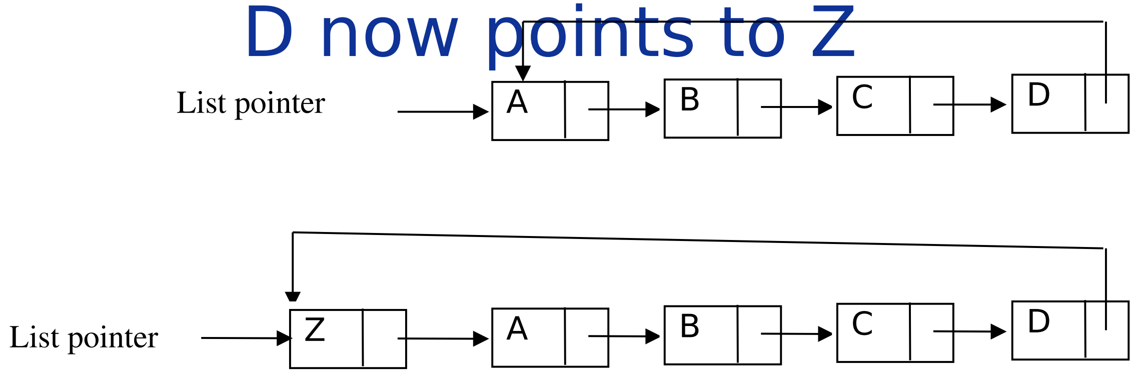
Adding nodes to the **middle** is easy...

Adding nodes to the **beginning** or **end** needs a different operation.

E.g., if we add Z to the start of the LL as we did before:

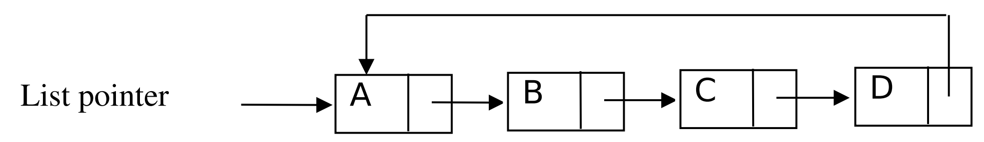
Z points to A

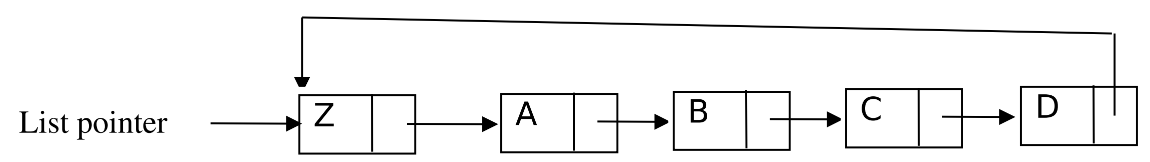
D now points to Z



Circular Linked-lists

Question: When printing the linked-list, how do you know you reached the end?



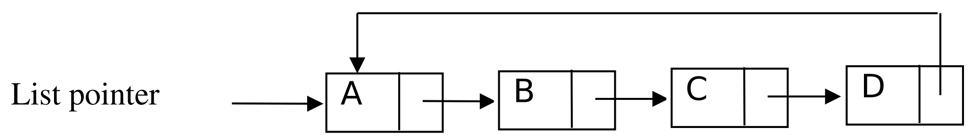


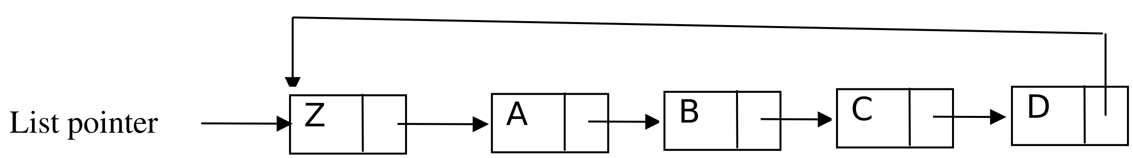
Te Kunenga ki Pūrehuroa

Circular Linked-lists

Question: When printing the linked-list, how do you know you reached the end?

Answer: `current->next == listpointer`





Te Kunenga ki Pūrehuroa

Circular Linked-lists

Exercise:

Modify the code for the function

```
void PrintLL(Node *listpointer)
```

So it prints a circular Linked-list without looping forever

Massey University
COLLEGE OF SCIENCES

Circular Linked-lists

Solution:

```
void CLL_PrintLL(Node *listpointer) {
// print all elements
Node *current;
  current = listpointer;
  int element=1;
  while (current->next != listpointer) {
        printf("Element %d: Balance of acc %i is %1.2f\n",
        element, current->accnumber, current->balance);
        current = current->next;
        element++;
  }
  //print last element
  printf("Element %d: Balance of acc %i is %1.2f\n",
  element, current->accnumber, current->balance);
  printf("End of the list.\n");
}
```

Te Kunenga ki Pūrehuroa

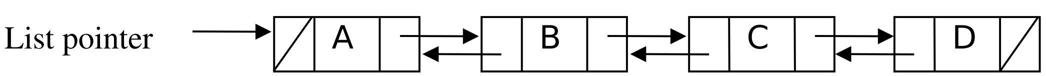
Doubly-linked Linked-lists

Have two pointers:

- Forward (next)
- Backward (previous)

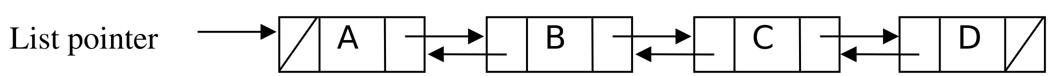
Advantages/disadvantages:

- Search backwards, deal with neighbours simultaneously
- More space in memory for the same amount of data
- Operations have two pointers to update



Doubly-linked Linked-lists

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



Massey University
COLLEGE OF SCIENCES

Challenge:

1) Can you think of a better method to reverse a linked-list, without any allocation or deallocation?

Te Kunenga
ki Pūrehuroa