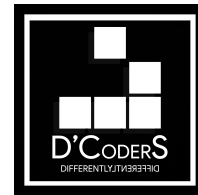


CLASSROOM SESSION



Loops in C

- Loops are used to repeat the execution of statement or blocks
- There are two types of loops
 - 1.Entry Controlled
For and While
 - 2. Exit Controlled
Do while

FOR Loop

FOR Loop has three parts: initialization, condition, increment

Syntax

```
for(initialization;condition;increment){  
  
    body;  
  
}
```

While Loop

It has a loop condition only which is tested to decide whether to execute the loop or not.

Syntax

```
while(<condition>){  
    Body;  
}
```

Do-While Loop

Do - While has a loop only condition only that is tested after each iteration to decide whether to continue with next iteration or terminate the loop.

Syntax:

```
do{  
    <body>  
}while(<condition>);
```

What Is Pattern?

A pattern is an arrangement of lines or shapes, especially a design in which the same shape is repeated at regular intervals over a surface.

Pattern Printing

*

* *

* * *

* * * *

* * * * *

Pattern Printing

*

* *

* * *

* * * *

* * * * *

Pattern Printing

*

* * *

* * * * *

* * * * * *

* * * * * *

Arrays

Outline

Introduction

Examples Using Arrays

Searching Using Arrays

Multidimensional Arrays

Introduction

- Arrays
 - Structure of related items
 - Static entry-same size throughout the program

An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array).

Declaring Arrays

When declaring arrays, specify

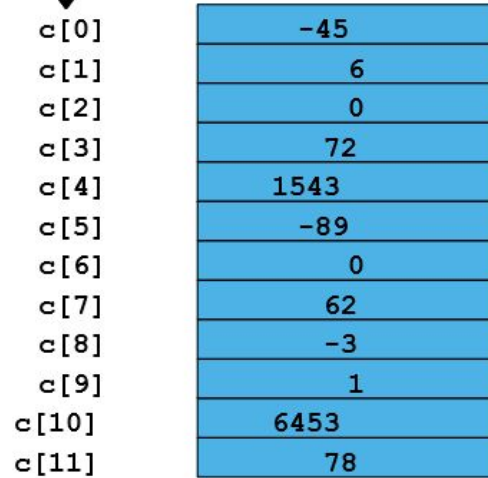
- Name
- Type of array
- Number of elements

arrayType arrayName[numberOfElements];

Examples:

- `int c[12];`
- `float myArray[3284];`

Name of array (Note
that all elements of
this array have the
same name, **c**)



c[0]	-45
c[1]	6
c[2]	0
c[3]	72
c[4]	1543
c[5]	-89
c[6]	0
c[7]	62
c[8]	-3
c[9]	1
c[10]	6453
c[11]	78

Position number of
the element within
array **c**

To refer to an element, specify

- Array name
- Position number

Format:

- arrayname[position number]
- First element at position 0
- n element array named c:
- c[0], c[1]...c[n - 1]
- **Array elements are like normal variables**

```
c[ 0 ] = 3;
```

```
printf( "%d", c[ 0 ] );
```

Initializers

- `int n[5] = { 1, 2, 3, 4, 5 };`

If not enough initializers, rightmost elements become 0

- `int n[5] = { 0 }`
- All elements 0
- C arrays have no bounds checking

If size omitted, initializers determine it

- `int n[] = { 1, 2, 3, 4, 5 };`

5 initializers, therefore 5 element array



Program to Find sum of all
elements of an array

Multi-Dimensional Array

Tables with rows and columns (m by n array)

Like matrices: specify row, then column Eg. `int arr[3][4]`

	Column 0	Column 1	Column 2	Column 3
Row 0	<code>a[0][0]</code>	<code>a[0][1]</code>	<code>a[0][2]</code>	<code>a[0][3]</code>
Row 1	<code>a[1][0]</code>	<code>a[1][1]</code>	<code>a[1][2]</code>	<code>a[1][3]</code>
Row 2	<code>a[2][0]</code>	<code>a[2][1]</code>	<code>a[2][2]</code>	<code>a[2][3]</code>

Diagram illustrating the structure of a 2D array (matrix) with 3 rows and 4 columns.

The array is represented as a table with rows and columns. The rows are labeled Row 0, Row 1, and Row 2. The columns are labeled Column 0, Column 1, Column 2, and Column 3.

The elements are accessed using the syntax `array_name[row_subscript][column_subscript]`.

Annotations:

- Array name: `a`
- Row subscript: The first subscript (e.g., `0` in `a[0][0]`).
- Column subscript: The second subscript (e.g., `0` in `a[0][0]`).

Initialization

- `int b[2][2] = { { 1, 2 }, { 3, 4 } };`
- Initializers grouped by row in braces
- If not enough, unspecified elements set to zero

1	2
3	4

`int b[2][2] = { { 1 }, { 3, 4 } };`

1	0
3	4

Referencing elements

- Specify row, then column

`printf("%d", b[0][1]);`



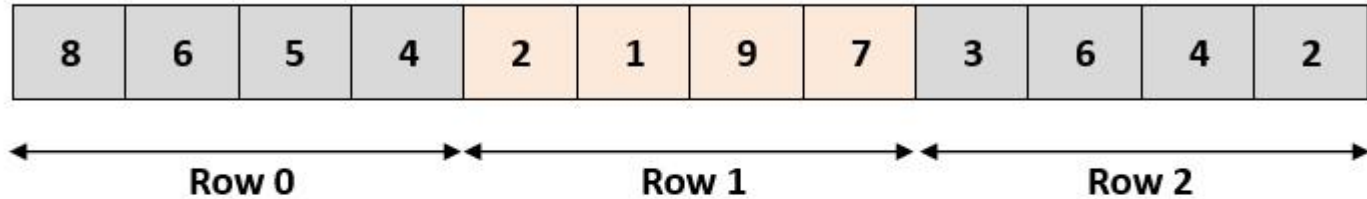
Program to print sum of main
diagonal of a 2D array

Storage of Multi-Dimensional Array

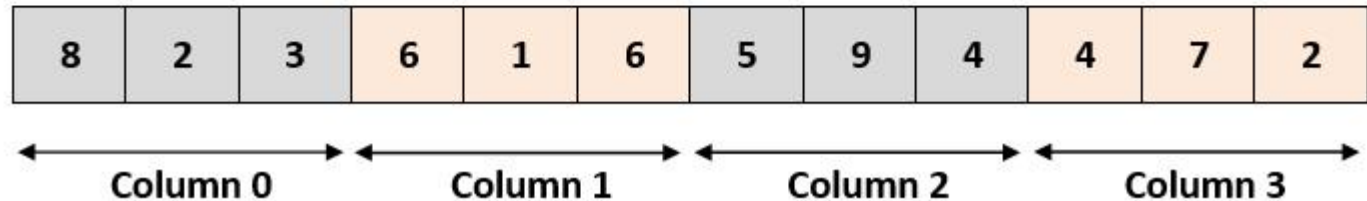
- **Row -Major Form**
- **Column -Major Form**
- **Neither Row-major or Column-major**

Eg. `int a[3][4]`

Row-Major (Row Wise Arrangement)



Column-Major (Column Wise Arrangement)



Every Programming Language implements its own method to store multidimensional arrays.

Values as stored in Memory:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----

Written down as

Column major: $\begin{pmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{pmatrix}$

Row major: $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix}$

Programming Languages

- **Row-Major**
 - C, C++, Objective-C, Pascal
- **Column-Major**
 - Fortran, MATLAB, R, Julia
- **Neither Row-Major nor Column-Major**
 - Java, Python

Special Case:

Row-Major order is the default in 'NumPy' library for Python

Pointer

A *pointer* is a reference to another variable (memory location) in a program

- Used to change variables inside a function (reference parameters)
- Used to remember a particular member of a group (such as an array)
- Used in dynamic (on-the-fly) memory allocation (especially of arrays)
- Used in building complex data structures (linked lists, stacks, queues, trees, etc.)

Outline

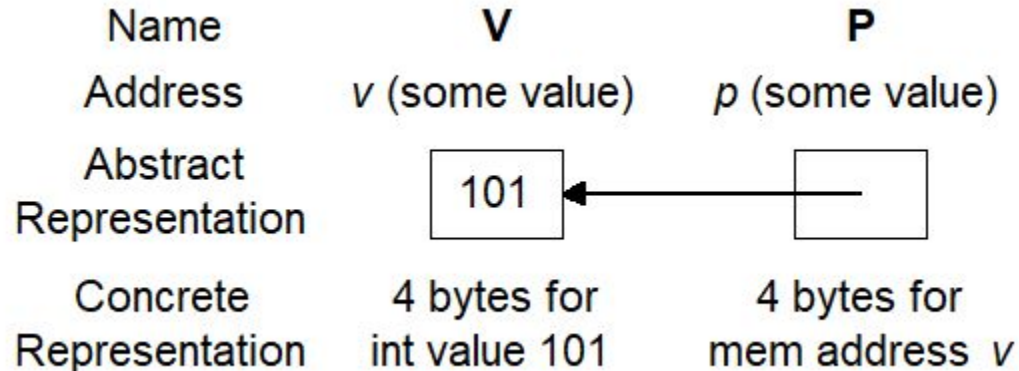
Variables are allocated at *addresses* in computer memory (address depends on computer/operating system)

Name of the variable is a reference to that memory address

A pointer variable contains a representation of an address of another variable (P is a pointer variable in the following):

```
int V = 101;
```

```
int *P = &V;
```



Pointer Variable Definition

Basic syntax: *Type *Name*

Examples:

```
int *P; /* P is var that can point to an int var */
```

```
float *Q; /* Q is a float pointer */
```

```
char *R; /* R is a char pointer */
```

Complex example:

```
int *AP[5]; /* AP is an array of 5 pointers to ints */
```

—more on how to read complex declarations later

Address (&) Operator

The address (&) operator can be used in front of any variable object in C -- the result of the operation is the location in memory of the variable

Syntax: *&VariableReference*

Examples:

int V;

int *P;

&V - memory location of integer variable V

&P - memory location of pointer variable P

Pointer Variable Initialization/Assignment

NULL - used to indicate pointer points to nothing

Can initialize/assign pointer vars to NULL or use the address (&) op to get address of a variable

–variable in the address operator must be of the right type for the pointer (an integer pointer points only at integer variables)

Examples:

```
int V;
```

```
int *P = &V;
```

```
int A[5]; P = &(A[2]);
```

Indirection (*) Operator

A pointer variable contains a memory address

To refer to the *contents* of the variable that the pointer points to, we use indirection operator

Syntax: **PointerVariable*

Example:

```
int V = 101;
```

```
int *P = &V;
```

```
/* Then *P would refer to the contents of the variable V (in this case, the integer 101) */
```

```
printf("%d",*P); /* Prints 101 */
```

Reference Parameters

To make changes to a variable that exist after a function ends, we pass the address of (a pointer to) the variable to the function (a reference parameter)

Then we use indirection operator inside the function to change the value the parameter points to:

```
void changeVar(float *cvar) {  
    *cvar = *cvar + 10.0;  
}  
  
float X = 5.0;  
changeVar(&X);  
printf("%.1f\n", X);
```

Pointer Return Values

A function can also return a pointer value:

```
float *findMax(float A[], int N) {  
    int I;  
    float *theMax = &(A[0]);  
    for (I = 1; I < N; I++)  
        if (A[I] > *theMax) theMax = &(A[I]);  
    return theMax;  
}
```

```
void main() {  
    float A[5] = {0.0, 3.0, 1.5,  
2.0, 4.1};  
    float *maxA;  
    maxA = findMax(A, 5);  
    *maxA = *maxA + 1.0;  
    printf("%.1f  
%.1f\n", *maxA, A[4]);  
}
```


Pointers to Pointers

A pointer can also be made to point to a pointer variable (but the pointer must be of a type that allows it to point to a pointer)

Example:

```
int V = 101;
```

```
int *P = &V; /* P points to int V */
```

```
int **Q = &P; /* Q points to int pointer P */
```

```
printf(“%d %d %d\n”,V,*P,**Q); /* prints 101 3 times */
```

1D Arrays and Pointers

`int A[5]` - `A` is the address where the array starts (first element), it is equivalent to `&(A[0])`

`A` is in some sense a pointer to an integer variable

To determine the address of `A[x]` use formula:

(address of `A` + `x` * bytes to represent `int`)

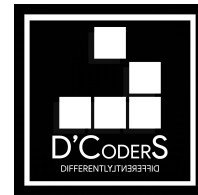
(address of array + element num * bytes for element size)

The `+` operator when applied to a pointer value uses the formula above:

`A + x` is equivalent to `&(A[x])` and `*(A + x)` is equivalent to `A[x]`

DYNAMIC MEMORY ALLOCATION (DMA)

CLASSROOM SESSION DAY 2



Arrays vs Linked List ?

1. Why not arrays ?
2. Why Linked List ?

ARRAYS

- Fixed Size. Cannot change the size once an array is declared.
- Random Access of Elements.
- Might result in unused space.
May lead to wastage of memory.

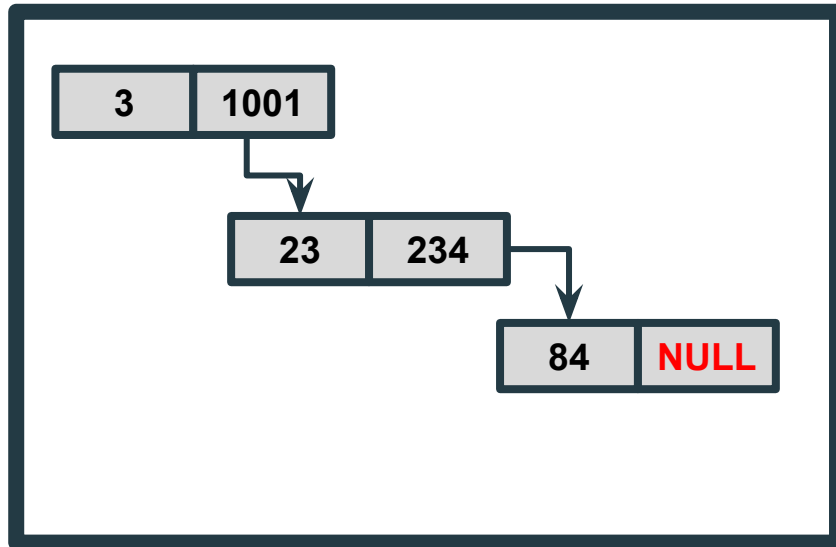
LINKED LISTS

- Dynamic Size.Can be changed as per convenience
- No Random Access of elements
- Dynamic Memory Allocation leads to no wastage of memory.



INTRODUCTION

A Linked List is a data structure whose size may change during its execution.

- Successive Elements are connected by pointers.
- Last element point to NULL.
- It can grow / shrink in size during the execution of the program.
- Doesn't have a predefined length.



Structure Declaration

```
struct Node {  
    int data;  2 bytes  
    struct Node * next;  2 bytes  
}
```


Contents

1. Array memory mapping
2. Structure memory mapping
3. Different insertion cases - explanation
4. Different insertion cases - coding'
5. Different deletion cases-explanation
6. Different deletion cases-Coding

ARRAY MEMORY MAPPING

Suppose this is your RAM which stores data for your program



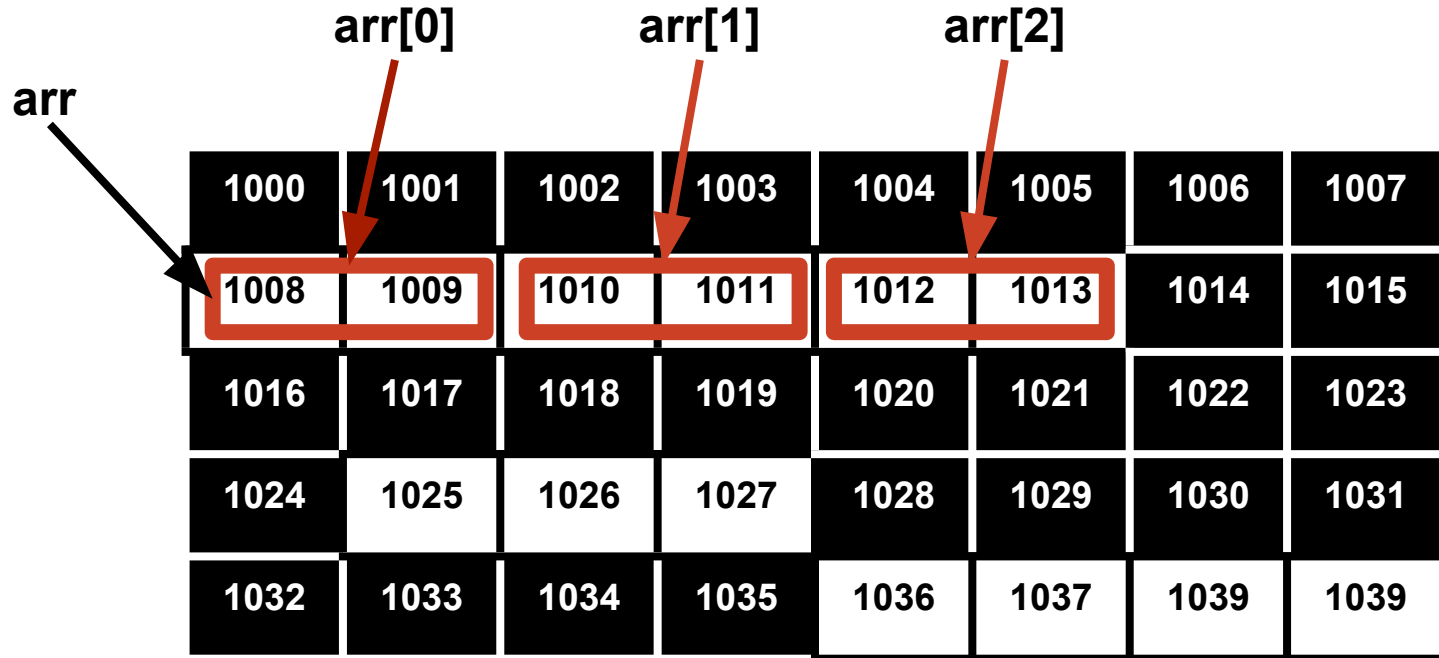
RAM

It is divided into many memory cells. Each memory cell is of 1 byte and has an address.

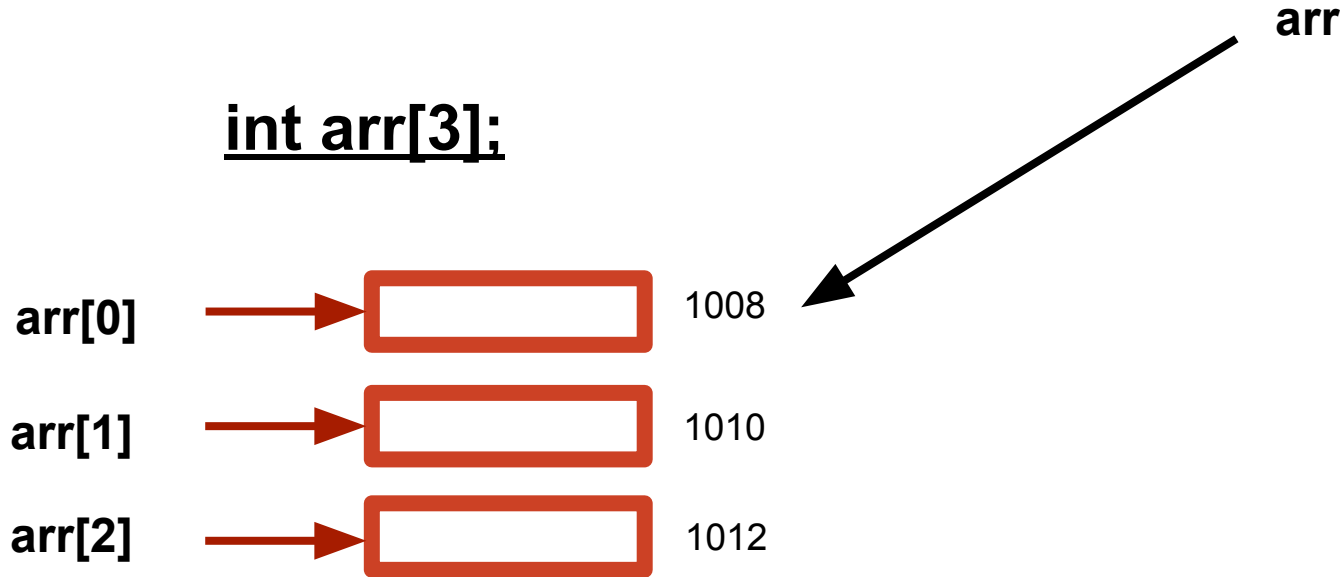
1000	1001	1002	1003	1004	1005	1006	1007
1008	1009	1010	1011	1012	1013	1014	1015
1016	1017	1018	1019	1020	1021	1022	1023
1024	1025	1026	1027	1028	1029	1030	1031
1032	1033	1034	1035	1036	1037	1039	1039

int arr[3];

(suppose size of integer to be of 2 bytes)



Simplifying memory mapping :



LINKED LIST MEMORY MAPPING

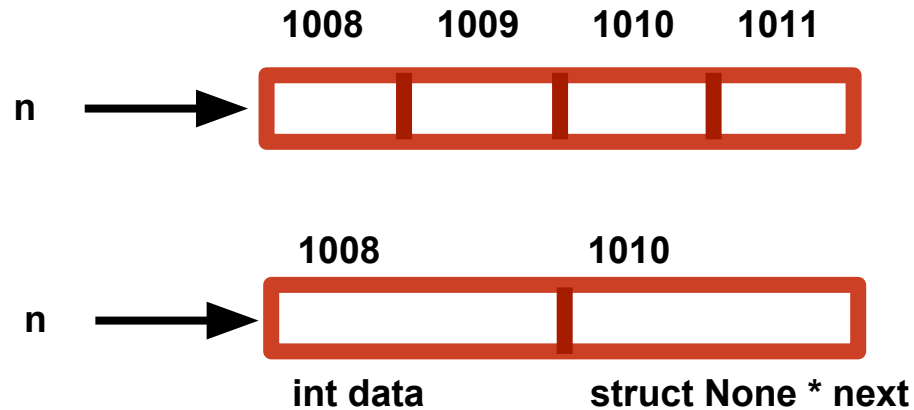
```
struct Node{  
    int data;  
    struct Node * next;  
}
```

int size = 2 bytes;
pointer size = 2 bytes;

```
struct Node * n = malloc( sizeof ( struct Node ) )
```

1000	1001	1002	1003	1004	1005	1006	1007
1008	1009	1010	1011	1012	1013	1014	1015
1016	1017	1018	1019	1020	1021	1022	1023
1024	1025	1026	1027	1028	1029	1030	1031
1032	1033	1034	1035	1036	1037	1039	1039

Simplifying memory mapping :



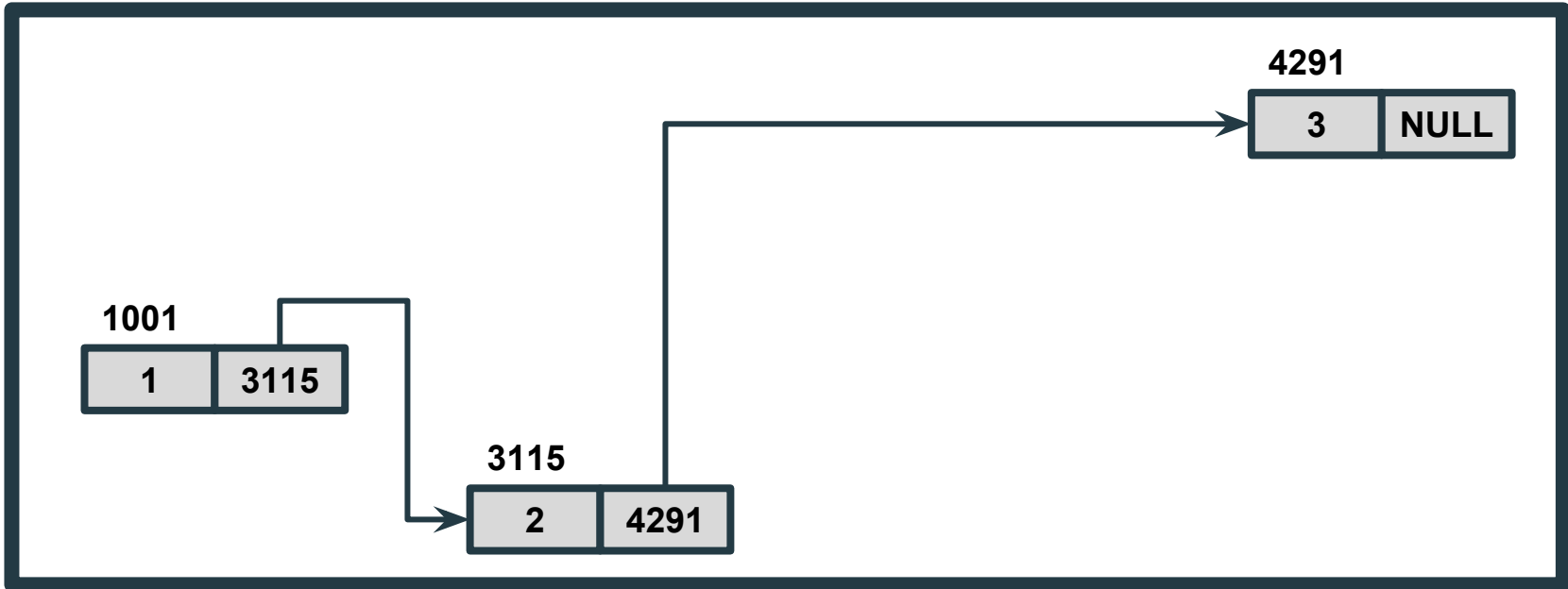
INSERTION

3 Different Types of Insertion

- 1) At the end of the linked list
- 2) At the end of the linked list.
- 3) After a given node.

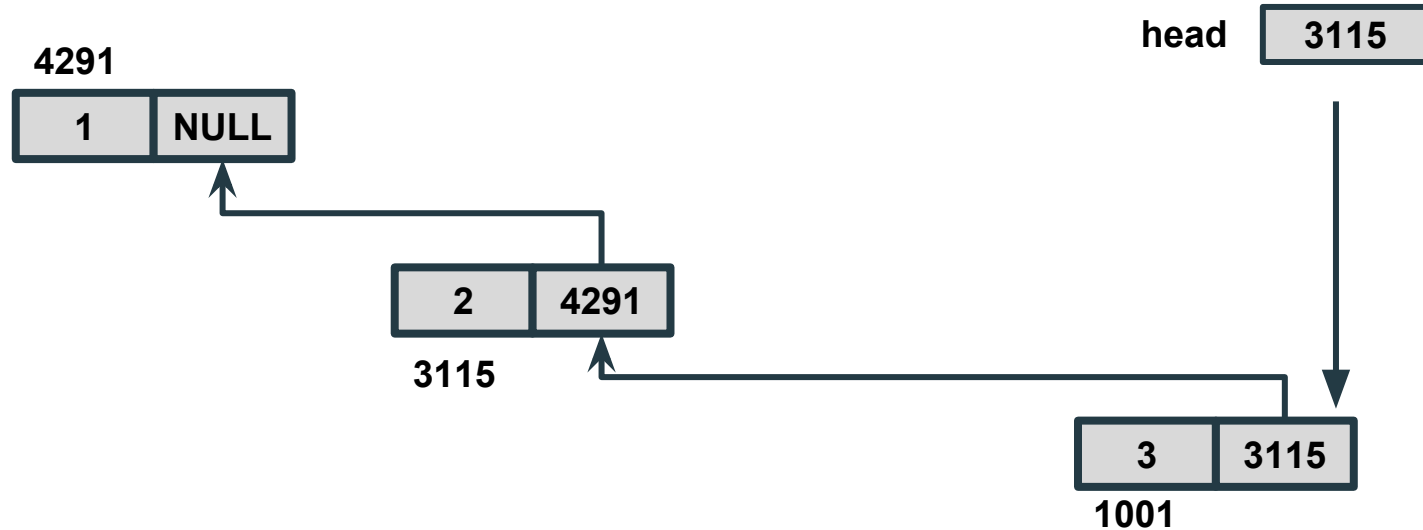
1) At the end of the linked list

Aim : to store three numbers 1,2,3



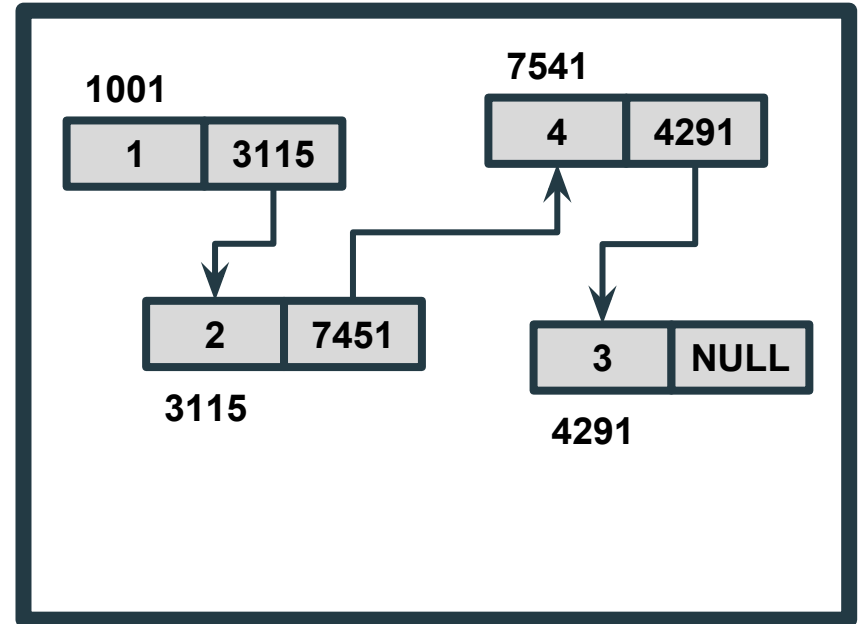
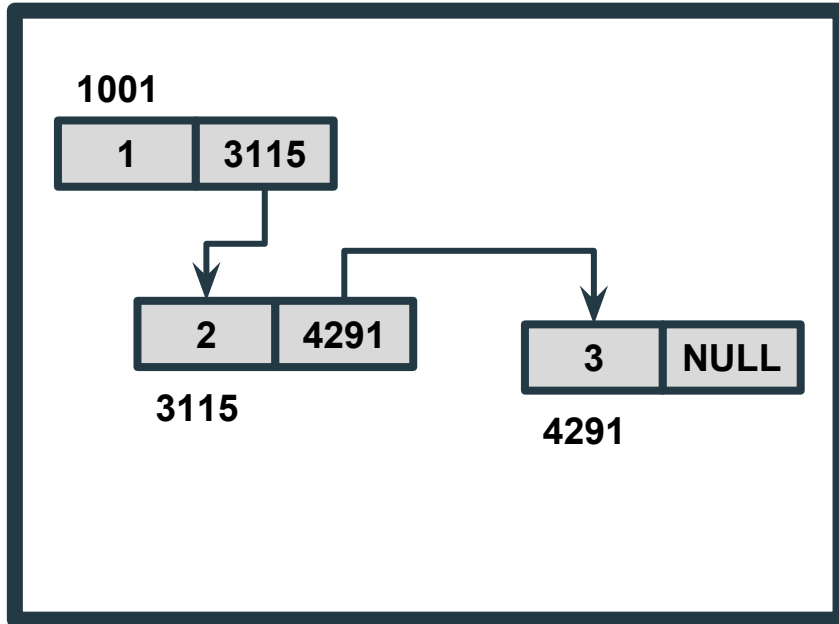
2) At the front of the linked list

Aim : to store three numbers 1,2,3



3) After a given node

Aim : to store 4 after index 1



**How to code insertion at
the end?**

Requirements

1. **head** pointer
 - a. Always points to the starting node
 - b. If no list empty then points to **NULL**
2. **temp** pointer
 - a. Used for iteration

Understanding insert() function

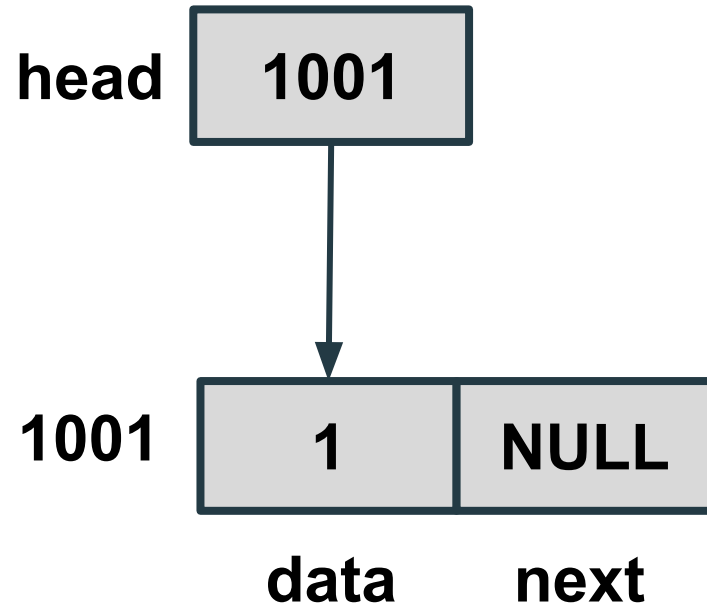
head

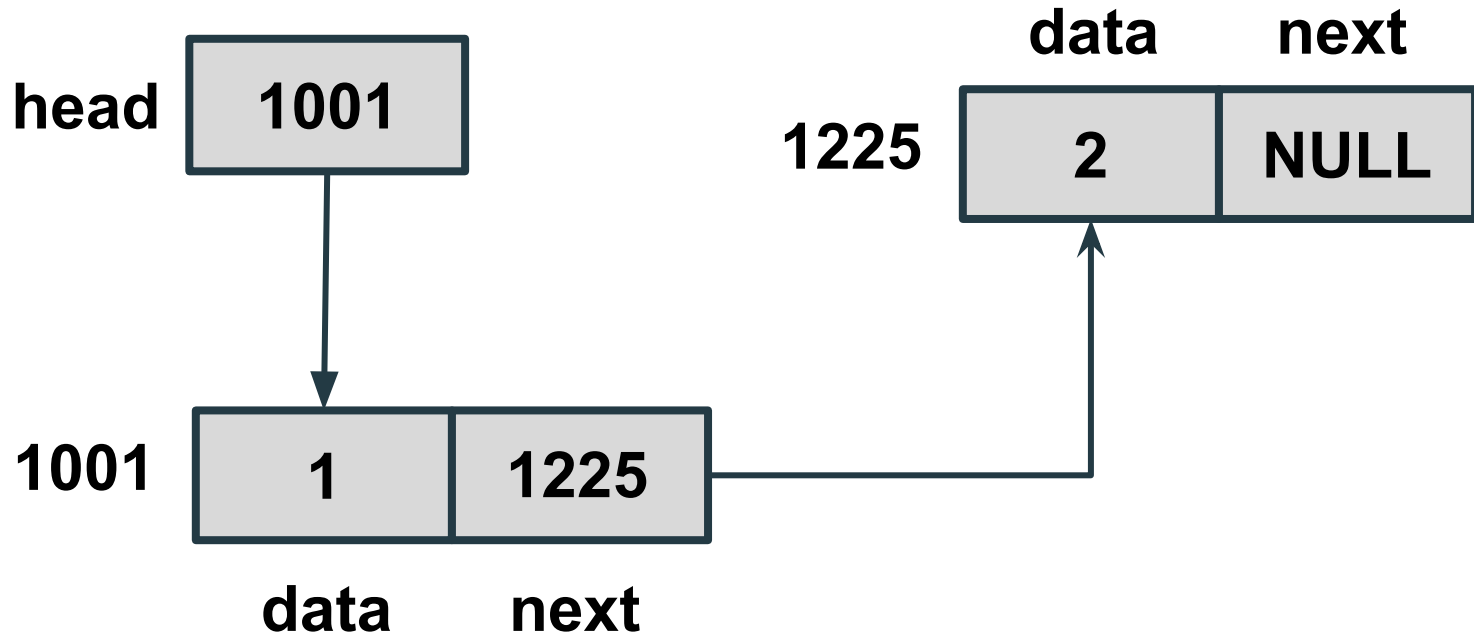


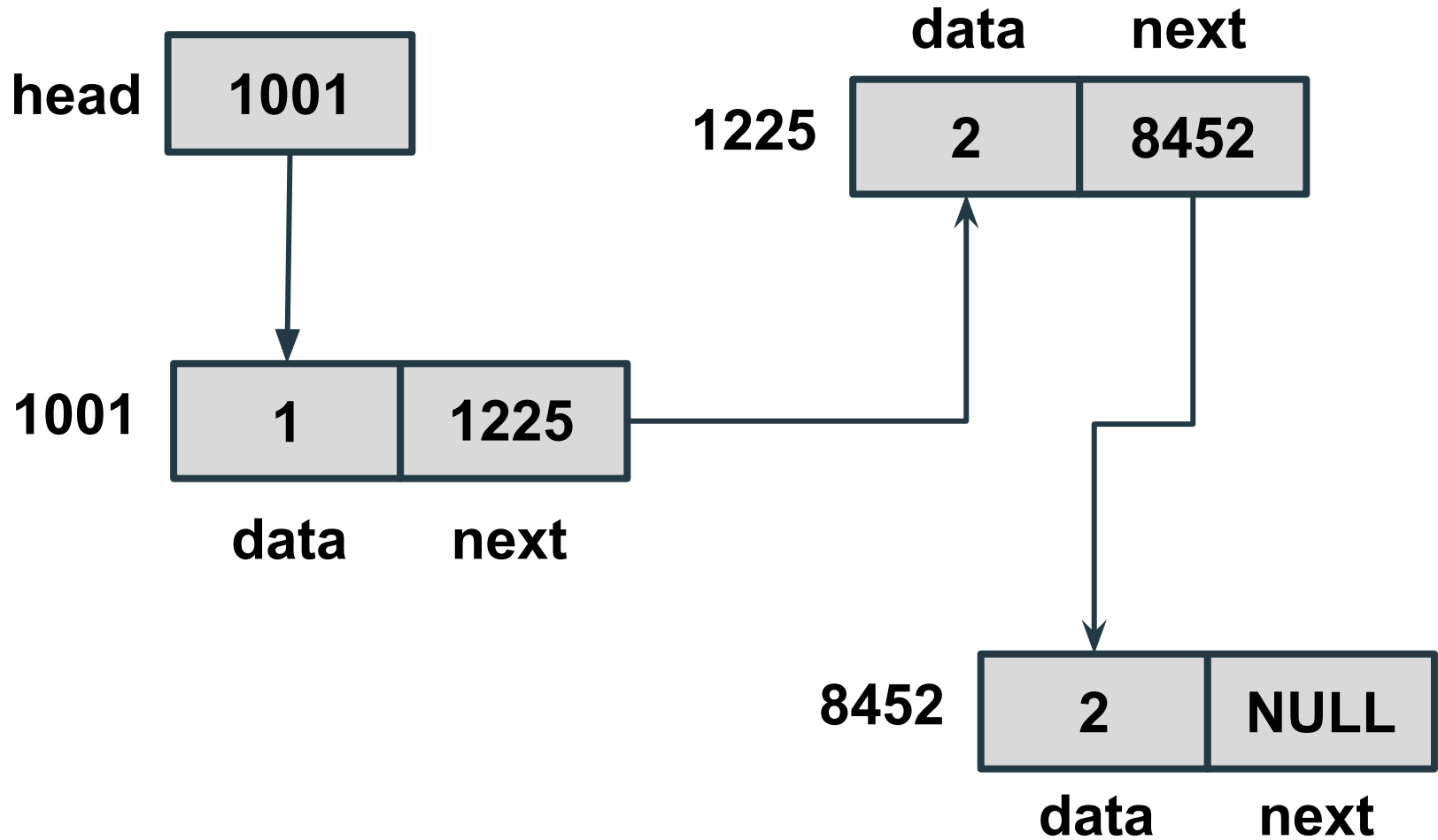
NULL

head

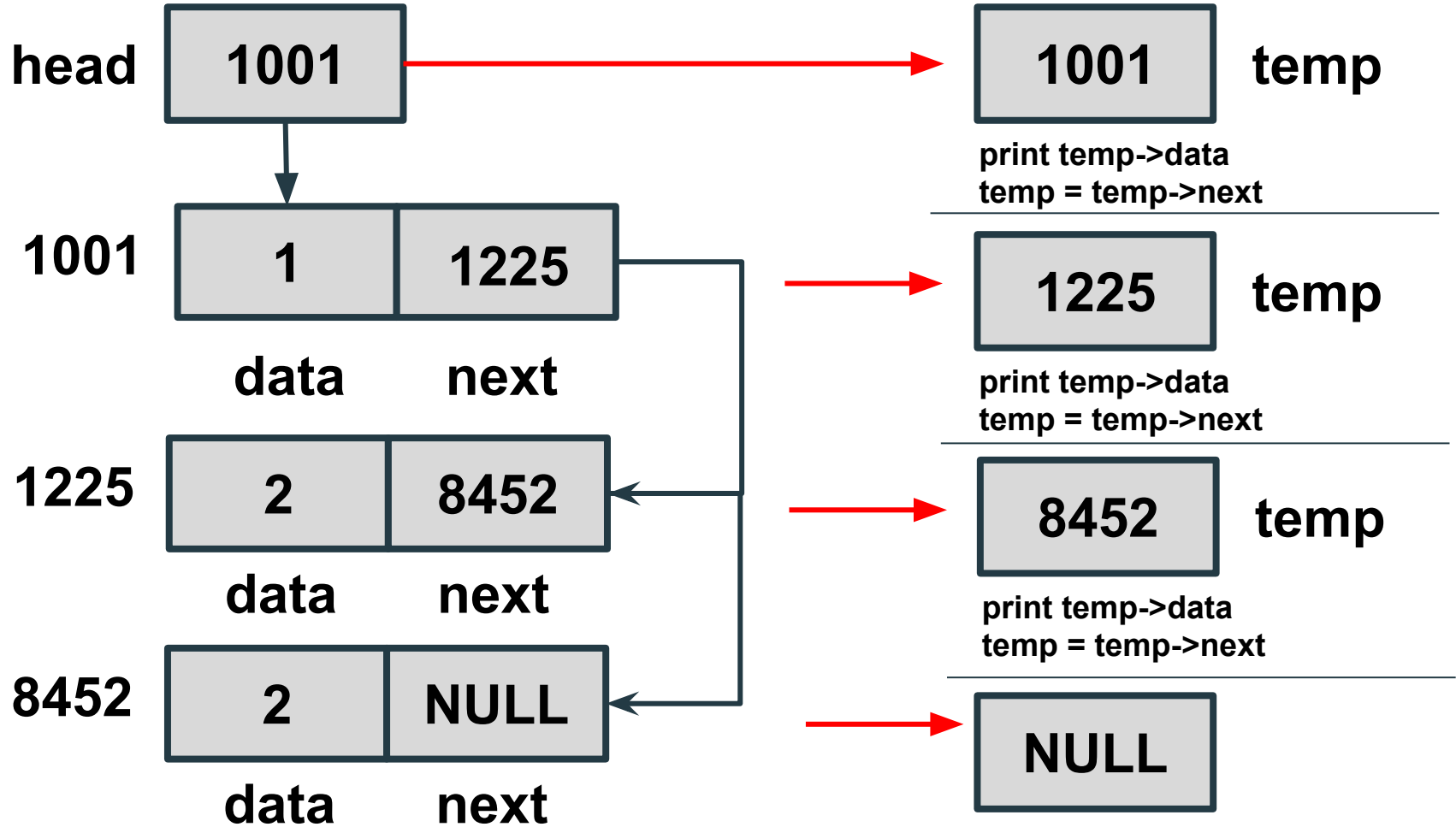
NULL







Understanding display() function



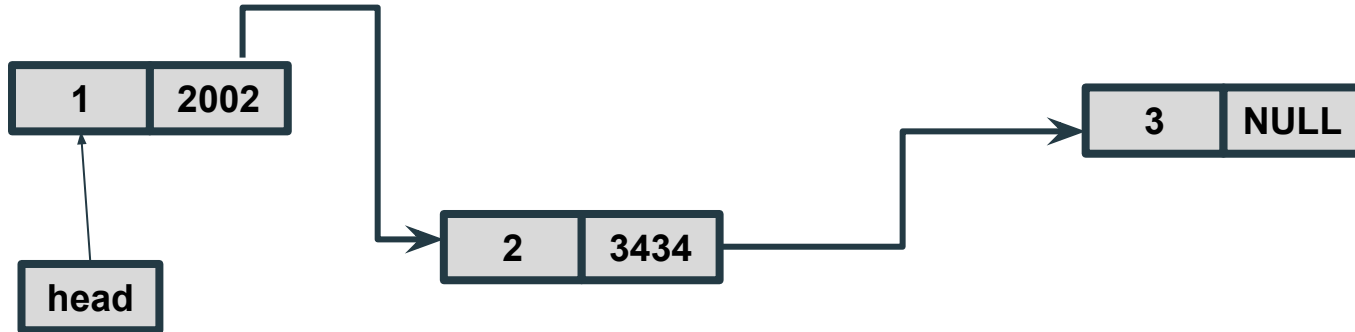
DELETION

3 Different Types of Deletion

1. At the beginning of the list.
2. After a given node.
3. At the end of the list

At the front of the list

Suppose you have a linked list like this ??



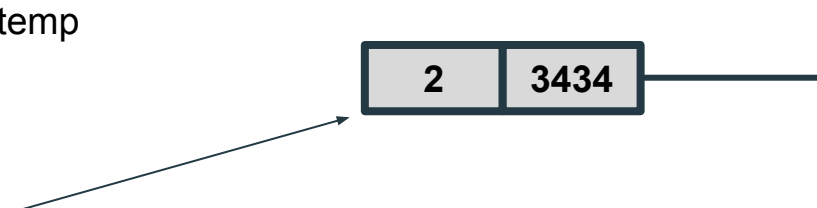
You want to delete node with data 1, i.e - the very first node.

At the front of the list

`free(temp)`



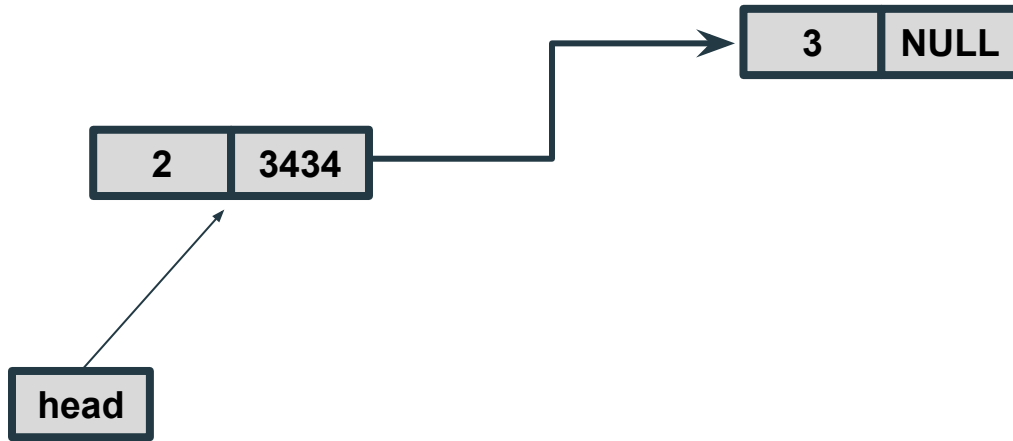
temp



Why free(temp) ?

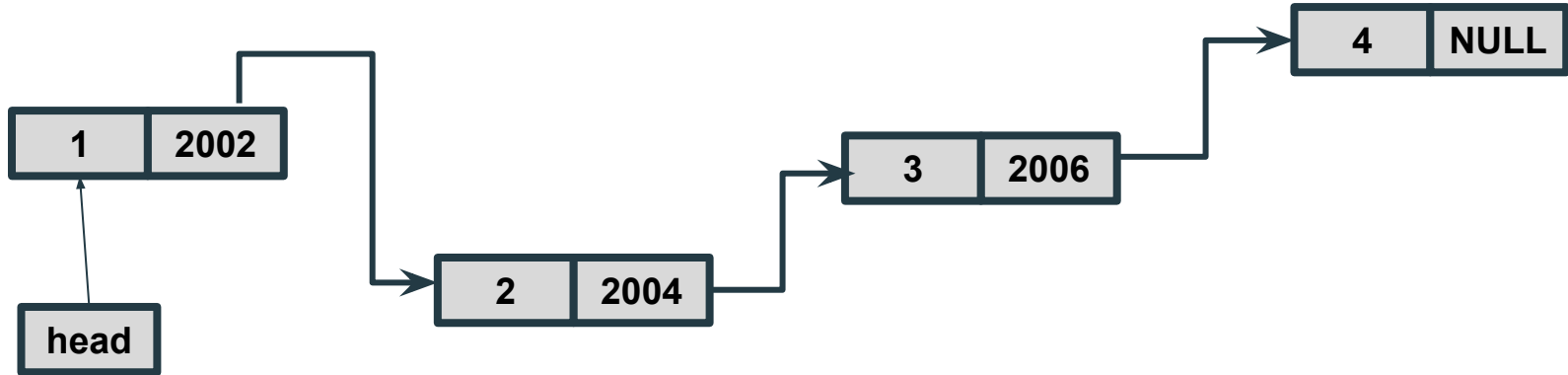
-free is a keyword in C.

Our new list



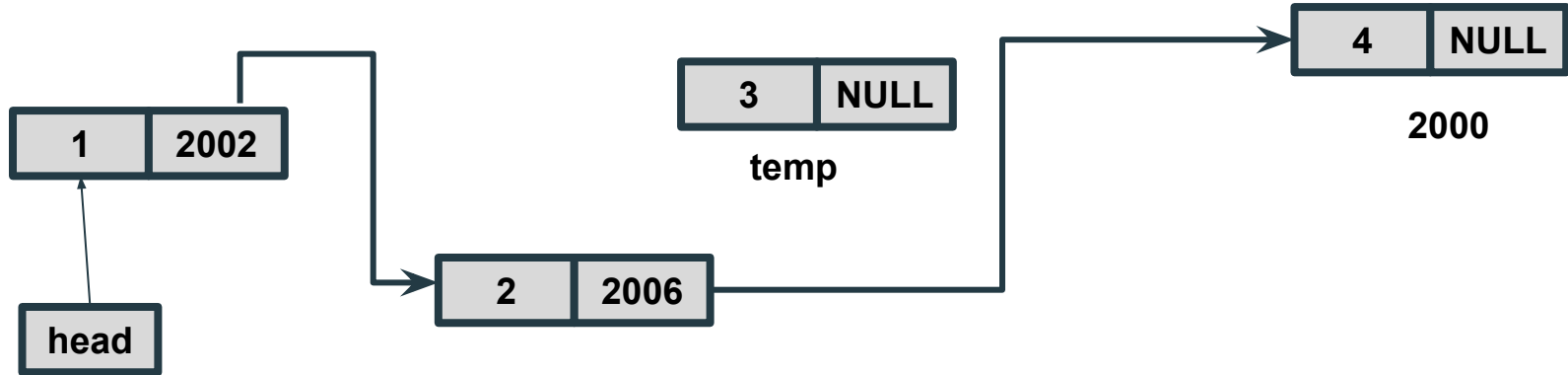
Given node of the list

Suppose you have a linked list like this ??



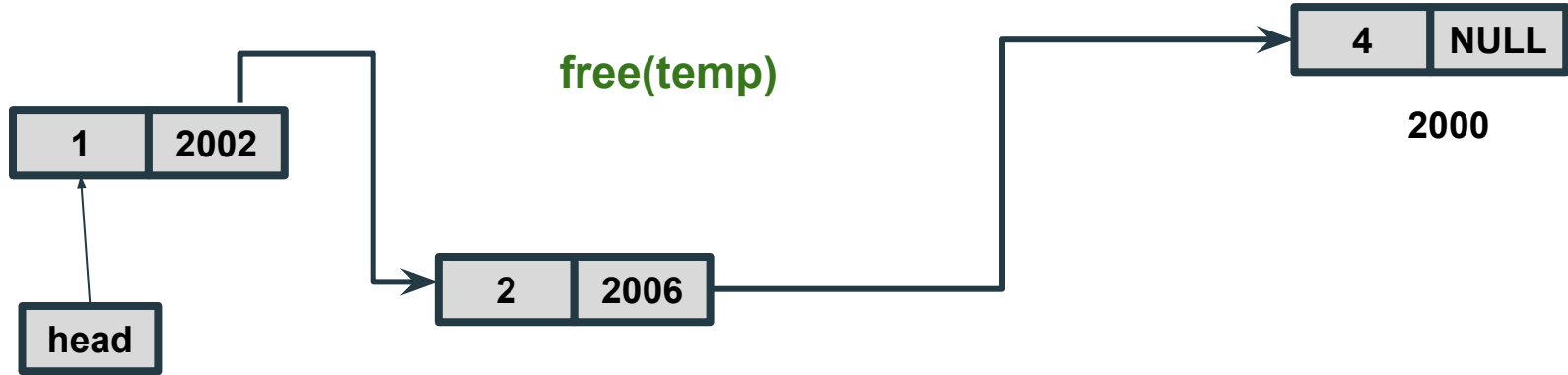
You want to delete node with data 3, i.e - the very first node.

Given node of the list



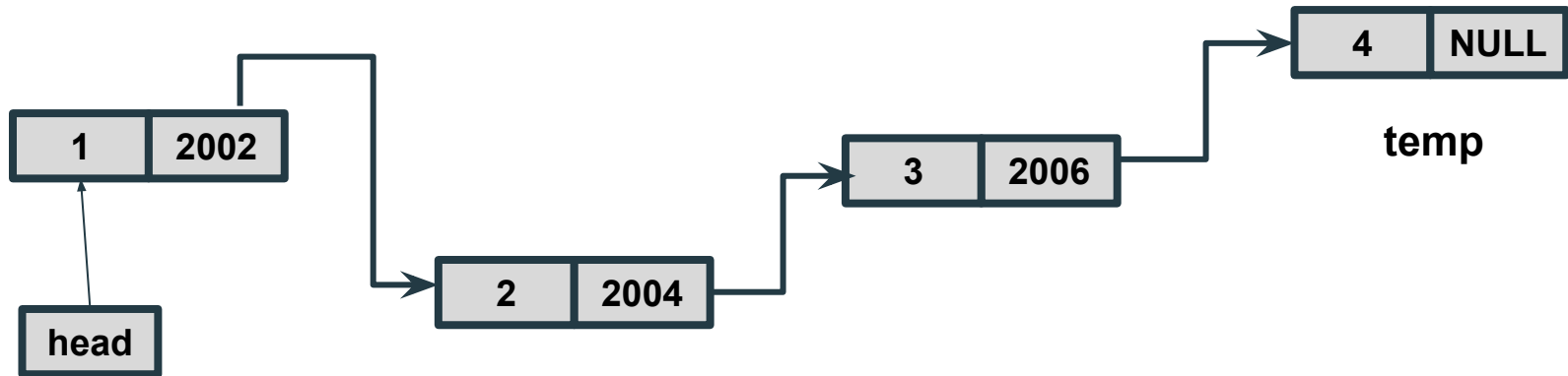
You want to delete node with data 3, i.e - the very first node.

Our new list



End of the list

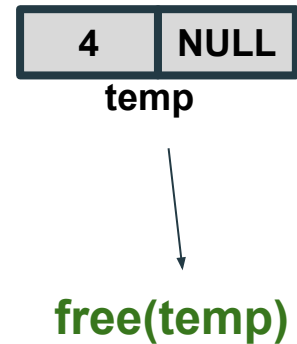
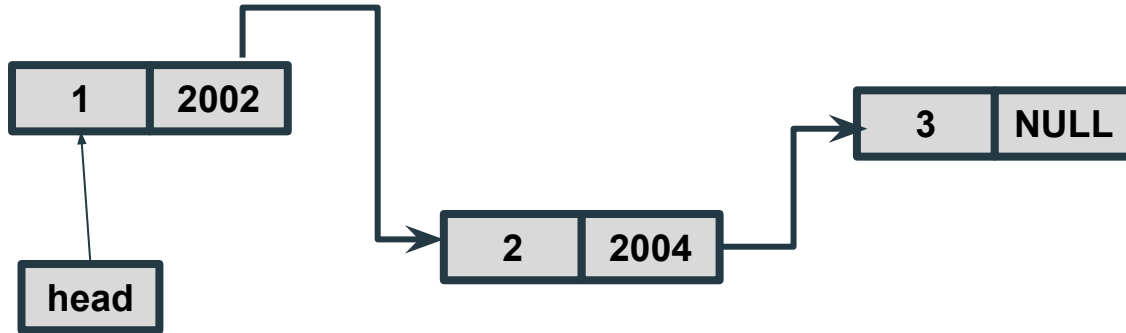
Suppose you have a linked list like this ??



You want to delete node with data 4, i.e - the last node.

End of the list

Suppose you have a linked list like this ??



You want to delete node with data 4, i.e - the last node.

Our new list

