Pointers

Pointer Introduction

- Pointer is a memory location or a variable which stores the memory address of another variable in memory.

- Pointers are more commonly used in C than other languages (such as BASIC, Pascal, and certainly Java, which has no pointers).

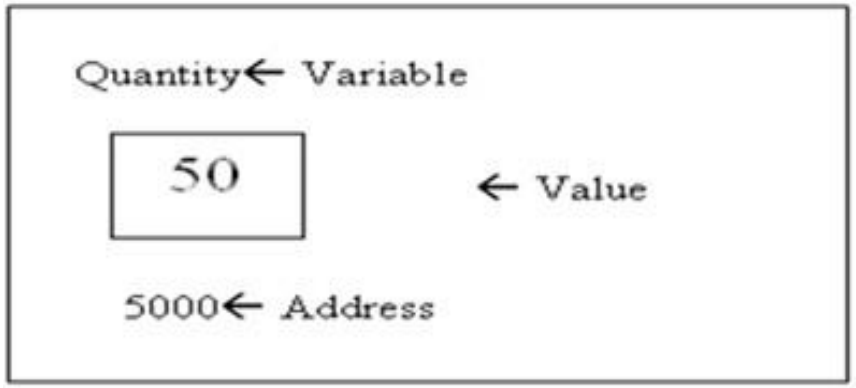
Common Uses of Pointers

- Accessing array elements.
- Passing arrays and strings to functions
- Passing arguments to a function when the function needs to modify the original argument.
- Obtaining memory from the system.
- Creating data structures such as linked lists.

Why Pointers?

- It enables the access to a variable that is defined outside the function.
- Pointers reduce the length and complexity of a program.
- They increase the execution speed.
- The use of a pointer array to character strings results in saving of data storage space in memory

Concept I



int Quantity = 50;

This statement instructs the system to find a location for the integer variable **Quantity** and puts the value **50** in that location.

- Assume that system has chosen address location 5000 for Quantity.

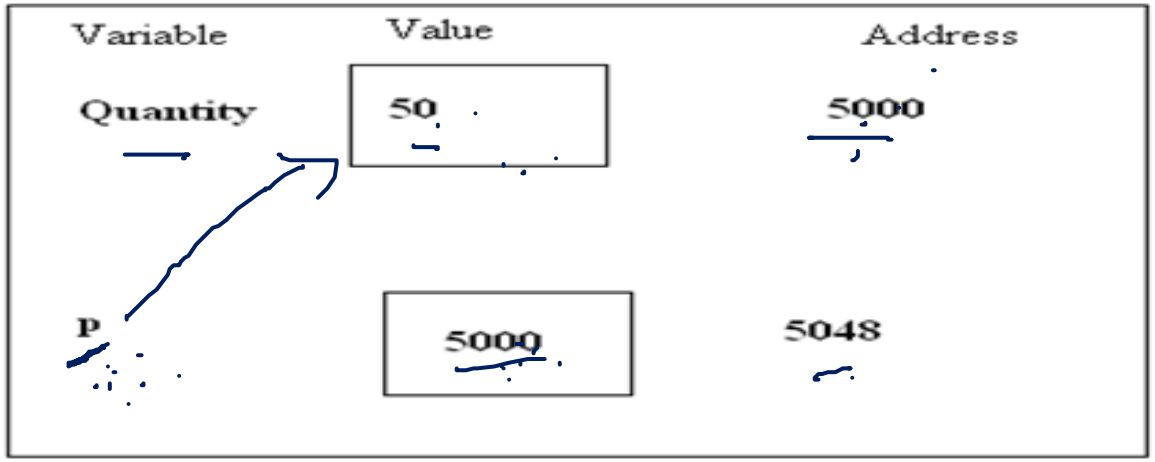
Concept II

- During Execution of the program, the system always associates the name Quantity with the address 5000.
- We may have access to the value 50 by using either the name of the variable Quantity or the address 5000.
- Since memory addresses are simply numbers, they can be assigned to some variables which can be stored in memory, like any other variable.
- To assign the address 5000 (the location of Quantity) to a variable p, we can write:

p = &Quantity ;

Such variables that hold memory addresses are called **Pointer Variables**

Concept III



The operator **&** can be called as **address of** and can be used only with a simple variable or an array element.

Declaring and Initializating Pointers I

- Declaration Syntax:
  <datatype> *pt_name;
- This tells the compiler 3 things about the pt name:
  - The asterisk(*) tells the variable pt_name is a pointer variable.
  - Pt_name needs a memory location.
  - Pt_name points to a variable of type data type
- Example:
  int *p; //declares a variable p as a pointer variable that points to an integer data type.
- float *x; // declares x as a pointer to floating point variable.

Declaring and Initializating Pointers II

- Once a pointer variable has been declared, it can be made to point to a variable using an assignment statement:

```
int quantity;
p = &quantity;
```

int *p;

int *p = &i;

- p now contains the address of quantity. This is known as **pointer initialization.**

To be taken care..

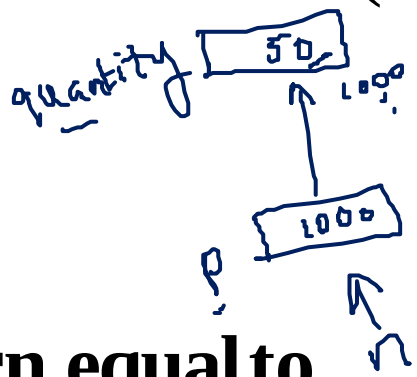
- Before a pointer is initialized, it should not be used.
- We must ensure that the pointer variables always point to the corresponding type of data.
- Assigning an absolute address to a pointer variable is prohibited.

  i.e p=5000
- A pointer variable can be initialized in its declaration itself.
- Example: **int x, *p=&x;** declares x as an integer variable and then initializes p to the address of x.
- **int *p = &x, x;** //not valid.

Accessing a variable through its pointer

- When the operator * is placed before a pointer variable in an expression on the R.H.S of equal sign, the pointer returns the value of the variable **quantity**
- The * can be remembered as value at address. (*p where p is the address!)
- Example:

```
p =&quantity;
n =*p; is equivalent to
n =*&quantity; which in turn equal to
n = quantity;
```

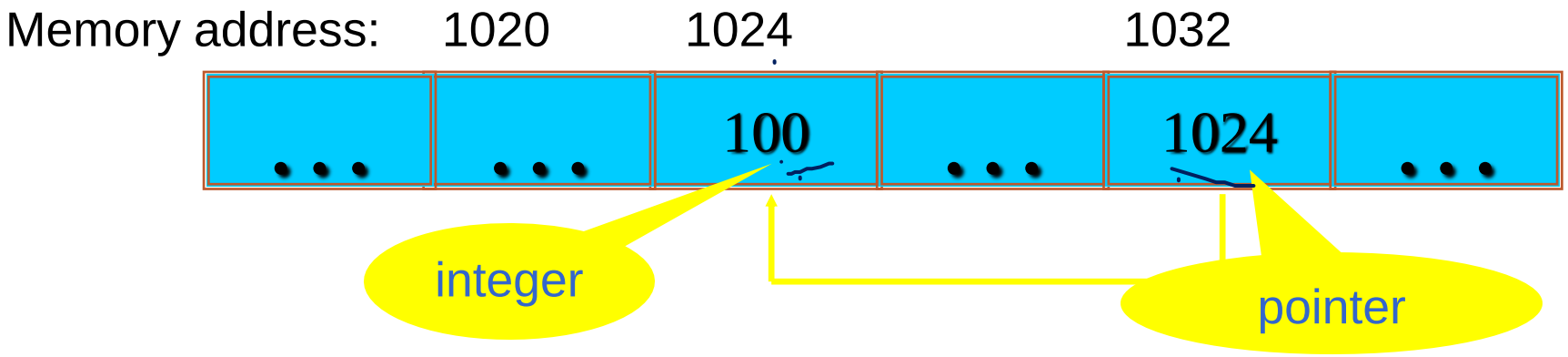


- & is the '**reference**' operator and can be read as "**address of**"
- * is the '**dereference**' operator and can be read as "**value pointed by**"

Pointers

A pointer is a variable used to store the address of a memory cell.

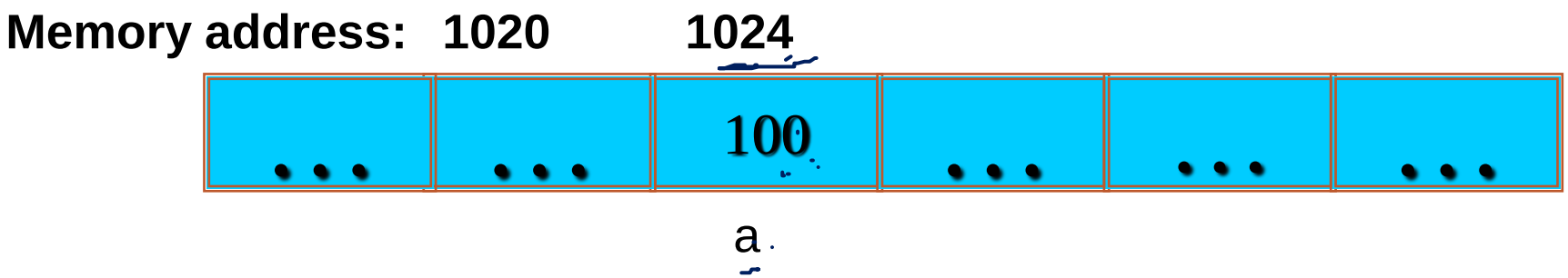
We can use the pointer to reference this memory cell



Address Operator &

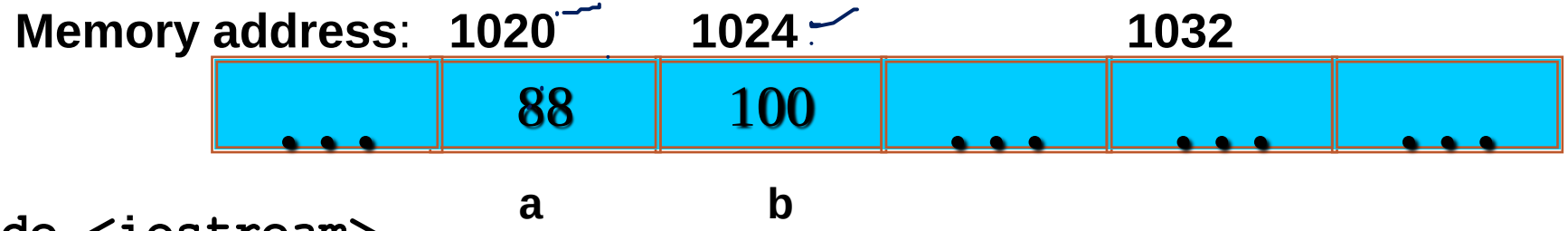
The "address of" operator (&) gives the memory address of the variable

- **Usage: `&variable_name`**




```
int a = 100;
//get the value,
cout << a;     //prints 100
//get the memory address
cout << &a;    //prints 1024
```

Address Operator &



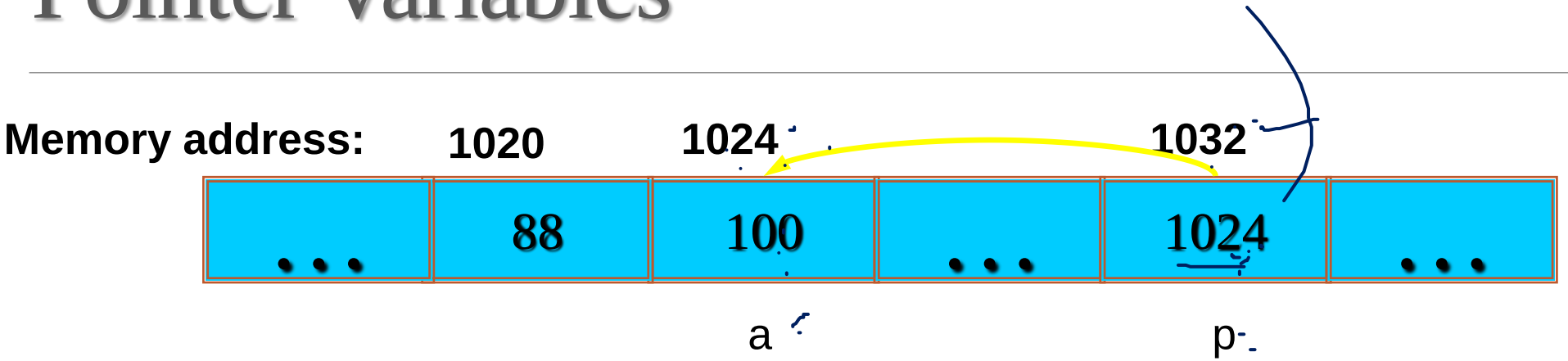

```
#include <iostream>
using namespace std;
void main(){
   int a, b;
   a = 88;
   b = 100;
   cout << "The address of a is: " << &a << endl;
   cout << "The address of b is: " << &b << endl;
}
```

- Result is:

The address of a is: 1020

The address of b is: 1024

Pointer Variables




```
int a = 100;
int *p = &a;
cout << a << " " << &a <<endl;
cout << p << " " << &p <<endl;
```

- Result is:

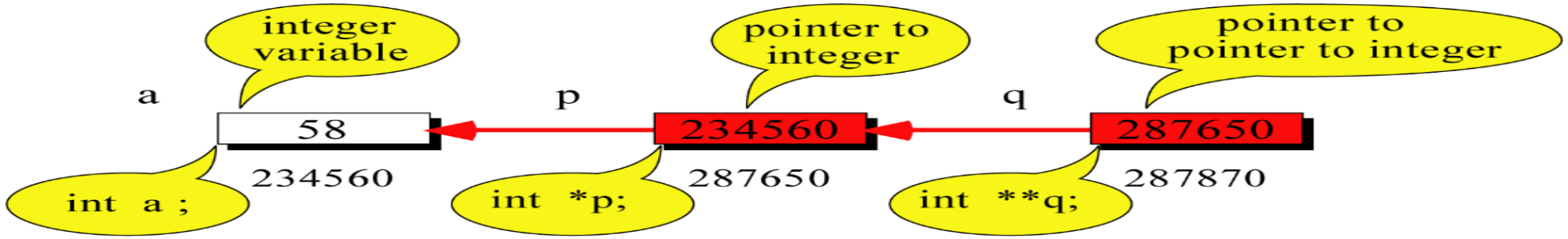
100 1024

1024 1032

- The value of pointer p is the address of variable a
- A pointer is also a variable, so it has its own memory address

Pointer to Pointer

```
// Local Declarations
int         a ;
int        *p ;
int       **q ;
```



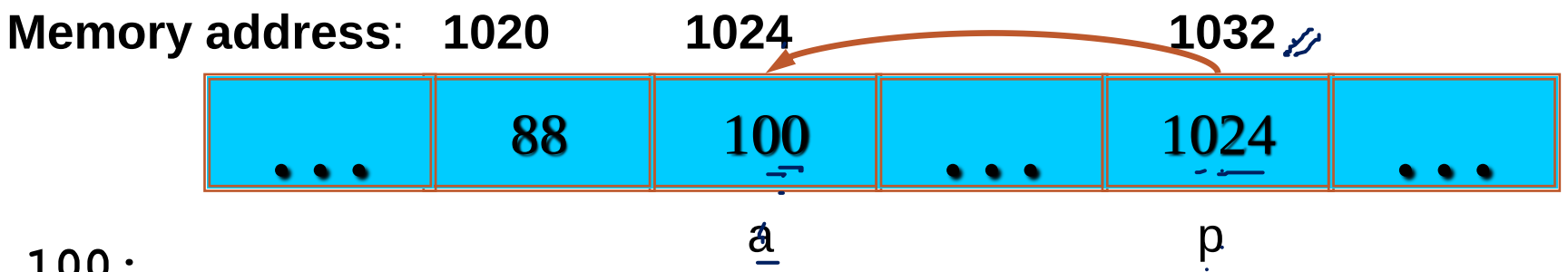
What is the output?

58 58 58

```
// Statements
a = 58 ;
p = &a ;
q = &p ;
cout <<       a << " ";
cout <<      *p << " ";
cout <<     **q << " ";
```

Dereferencing Operator *

We can access to the value stored in the variable pointed to by using the dereferencing operator (*),




```
int a = 100;
int *p = &a;
cout << a << endl;
cout << &a << endl;
cout << p << " " << *p << endl;
cout << &p << endl;
```

- Result is:

100
1024
1024 100
1032

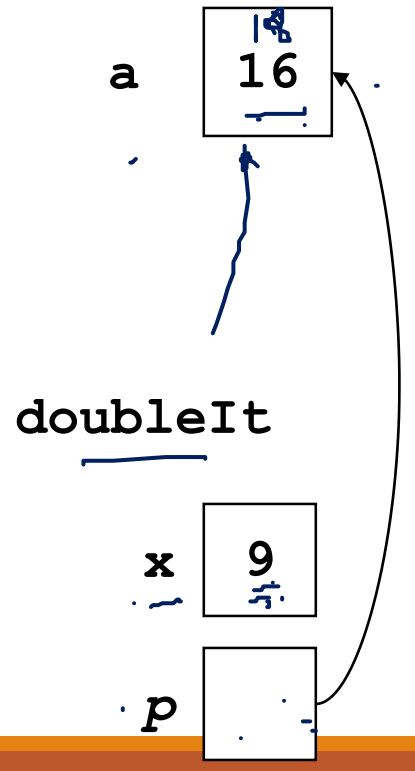
A Pointer Example

Output=a gets 18

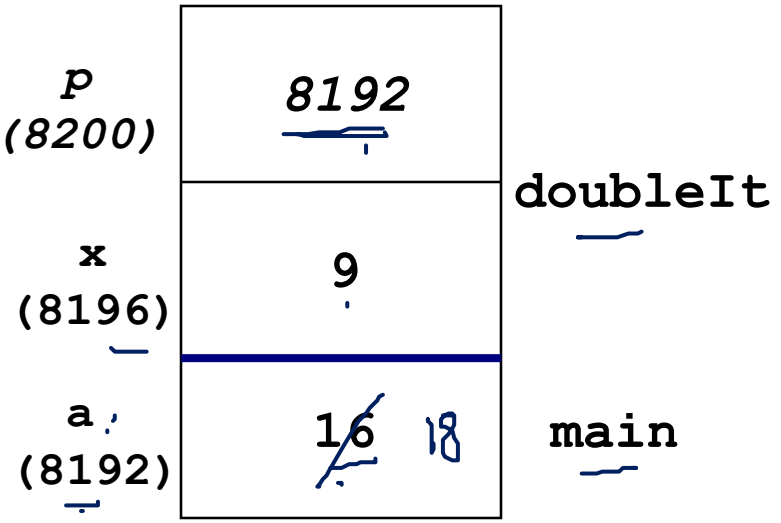
The code

```
void doubleIt(int x, int * p)
{
 *p = 2 * x;
}

int main()
{
 int a = 16;
 doubleIt(9, &a);
 return 0;
}
```

Box diagram

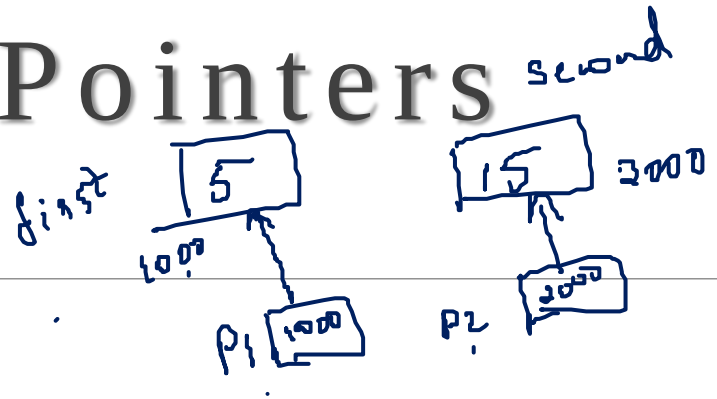


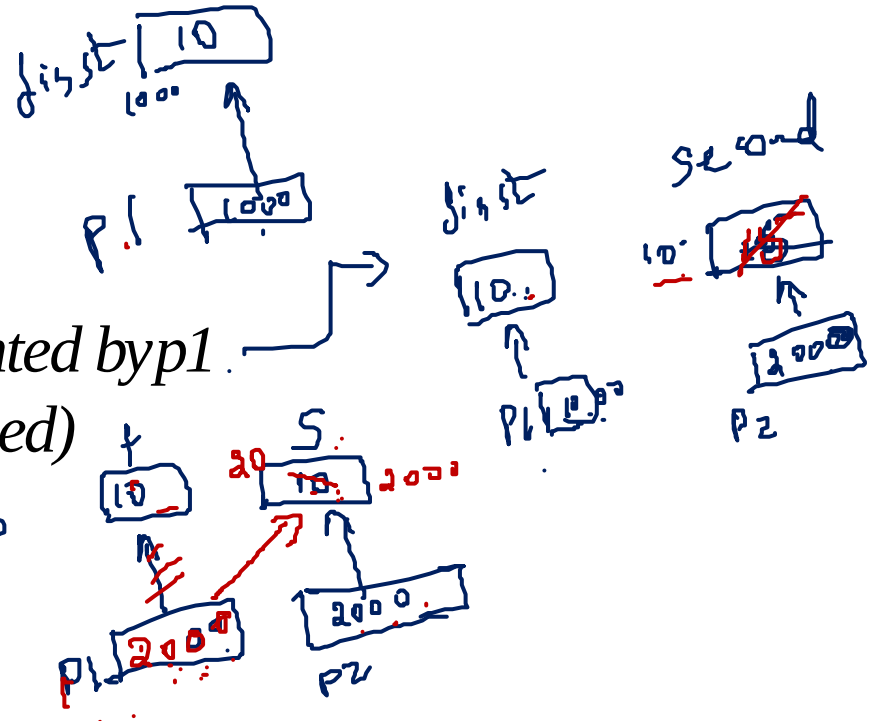
Memory Layout



Understanding Pointers

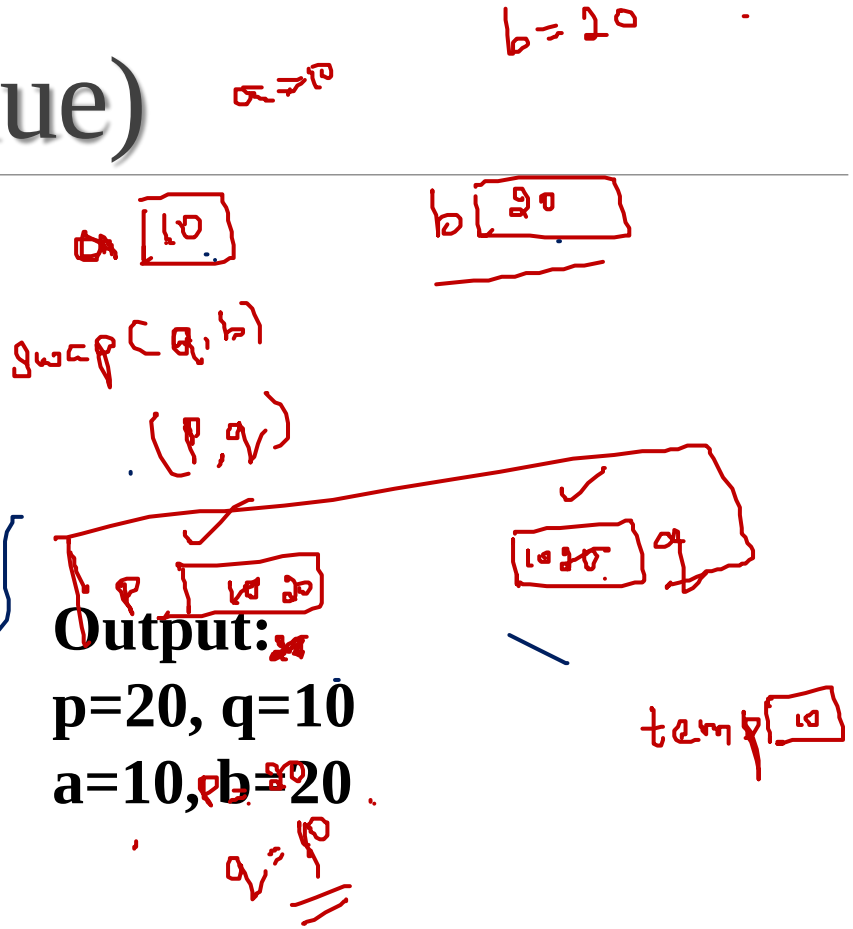
```
int main ()
{
  int firstvalue = 5, secondvalue = 15;
  int * p1, *p2;
  p1 = &firstvalue;          // p1 = address of firstvalue
  p2 = &secondvalue;         // p2 = address of secondvalue
  *p1 = 10;                  // value pointed by p1 = 10
  *p2 = *p1;                 // value pointed by p2 = value pointed by p1
  p1 = p2;                   // p1 = p2 (value of pointer is copied)
  *p1 = 20;                  // value pointed by p1 = 20
  cout << "firstvalue is " << firstvalue;
  cout << "secondvalue is " << secondvalue;
  return 0;
}
```





Pointer Usage (pass by value)

```
void IndirectSwap(int p, int q){
   int temp = p;
    p = q;
    q= temp;
   cout << p << q << endl;
}

int main() {
   int a = 10;
   int b = 20;
   IndirectSwap(a, b);
   cout << a << b << endl;
   return 0;
}
```



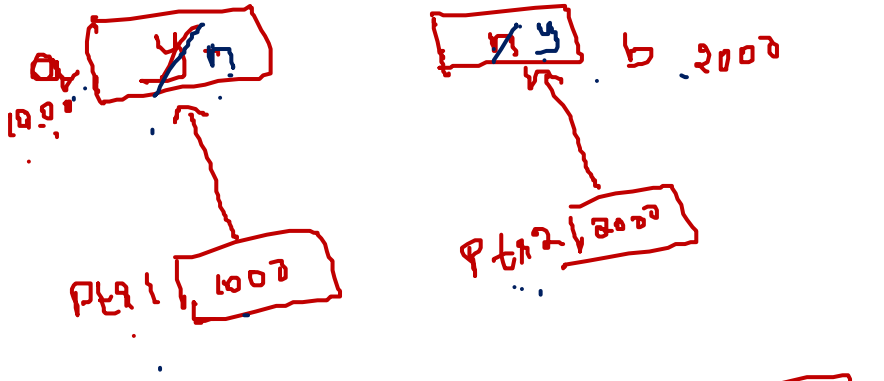
Output:

p=20, q=10

a=10,b=20

Pointer Usage (pass by address)

```
void IndirectSwap(char *Ptr1, char *Ptr2){
  char temp = *Ptr1;
  *Ptr1 = *Ptr2;
  *Ptr2 = temp;
}

int main() {
  char a = 'y';
  char b = 'n';
  IndirectSwap(&a, &b);
  cout << a << b << endl;
  return 0;
}
```



Output:

a=n

b=y

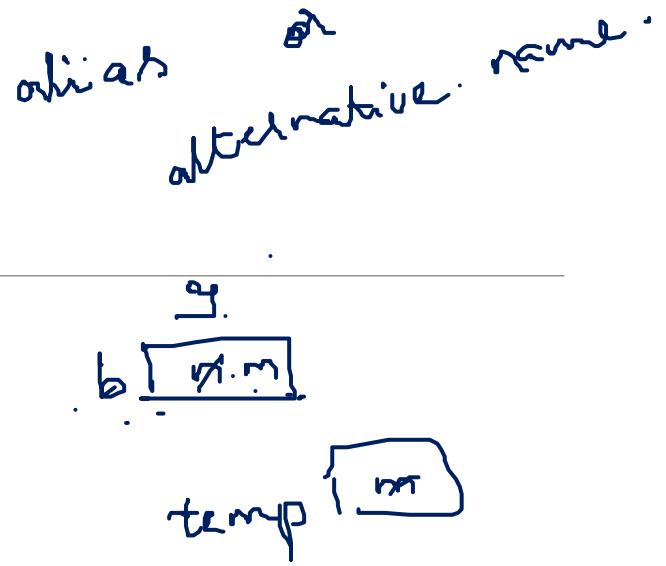
Pass by Reference (only in C++)

```
void IndirectSwap(char & x, char & y) {
   char temp = x;
   x = y;
   y = temp; }

int main() {
   char a = 'm';
   char b = 'n';
   IndirectSwap(a, b);
   cout << a << b << endl;
   return 0;
}
```

Output:

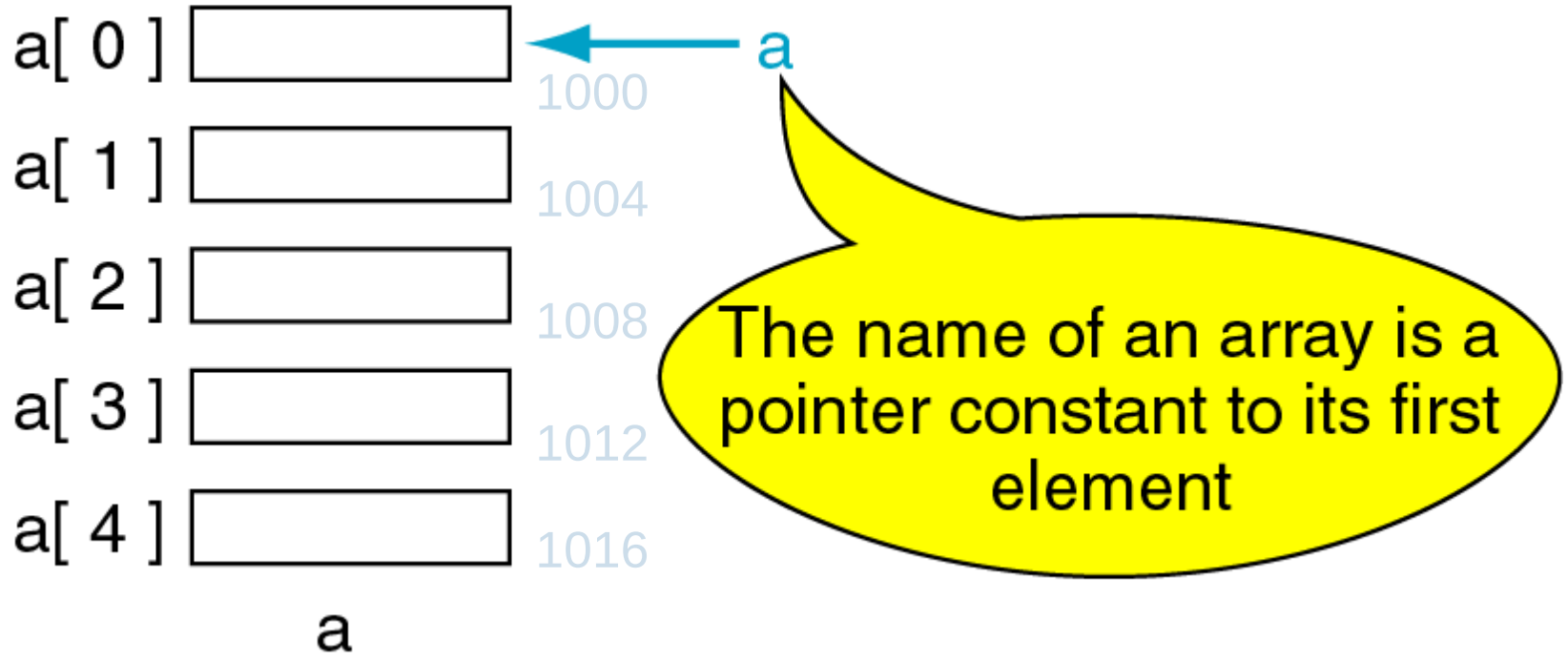
a=n

b=m



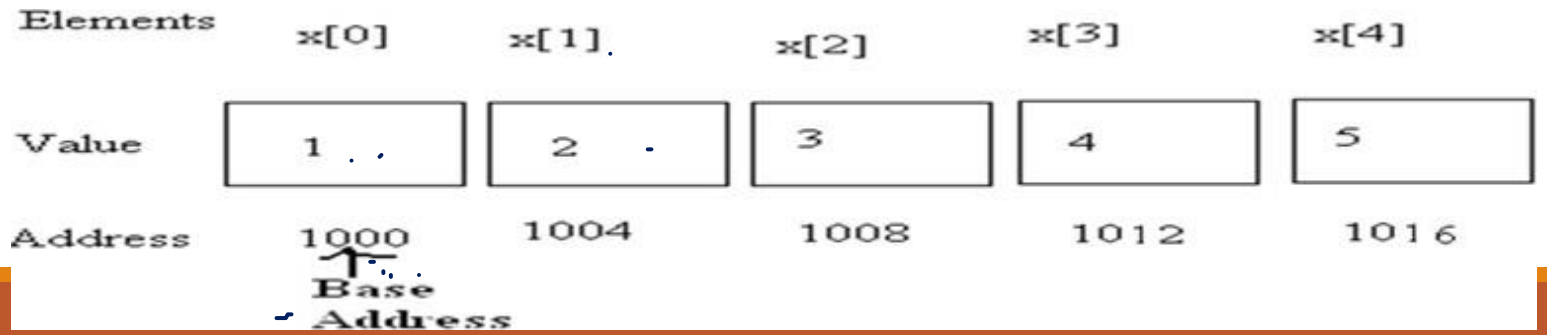
Pointers and Arrays

- The name of an array points only to the first element not the whole array.



Pointers and Arrays I

- When an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations.
- An array x is declared as follows and assume the base address of x is 1000.
  int x[5] ={1,2,3,4,5};
- Array x, is a constant pointer, pointing to the first element **x[0]**. Value of **x** is **1000** (Base Address), the location of x[0] i.e. **x = &x[0] = 1000**



Pointers and Arrays II (pointer to array)

- Use a **pointer to an array**, and then use that pointer to access the array elements.
- An integer pointer variable p, can be made to point to an array as follows:

```
int x[5] = { 1,2,3,4,5};
int *p;
p =  x; OR p =  &x[0];

p =  &x ; //Invalid
```

- Successive array elements can be accessed by writing:

```
cout<< *p; p++;   or
cout <<*(p+i); i++;
```

```c
#include<stdio.h>
 void main()
{
 int x[3] = {7, 9, 45};
 int *p = x;
 for (int i = 0; i < 3; i++)
 { printf("%d", *p);
    p++;
 }
 return 0;
 }
```

*(p+i)//pointer with an array is same as x[i]

Pointers and Arrays III

- Address of an element of x is given by:
  Address of x[i] = base address + i * scale factor of (int)
  Address of x[3]= 1000 +(3*2) = 1006
- The relationship between p and x is shown below.

  p = &x[0] (=1000)
  p+1 = &x[1] (=1002)
  p+2 = &x[2] =1004)
  p+3 = &x[3] (=1006)
  p+4 = &x[4]
- ***(p+3) is equivalent to x[3].**

Array Name is a pointer constant

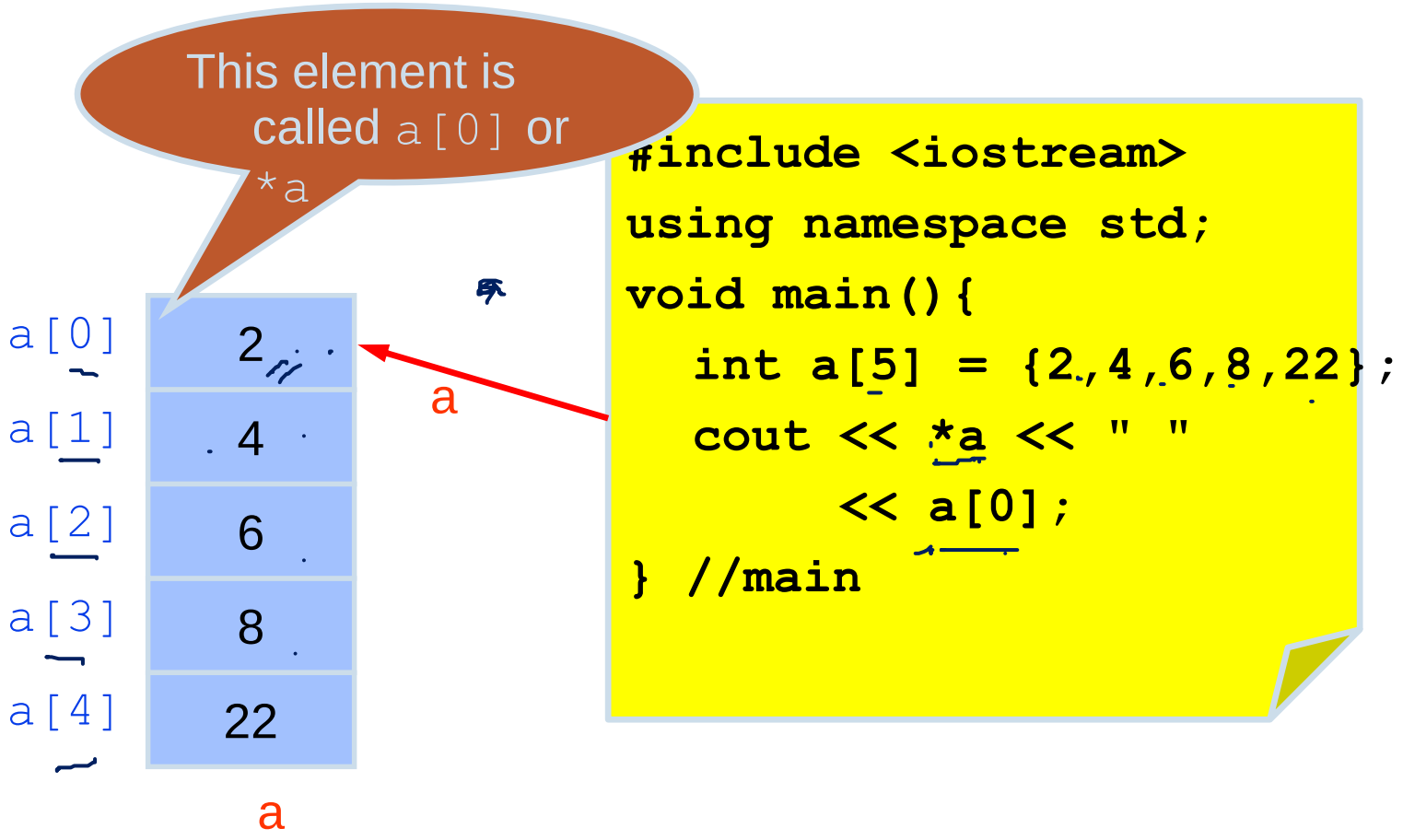
```
#include <iostream>
using namespace std;

void main (){
    int a[5];
    cout << "Address of a[0]: " << &a[0] << endl
         << "Name as pointer: " << a << endl;
}
```

Result:

Address of a[0]: 0x0065FDE4

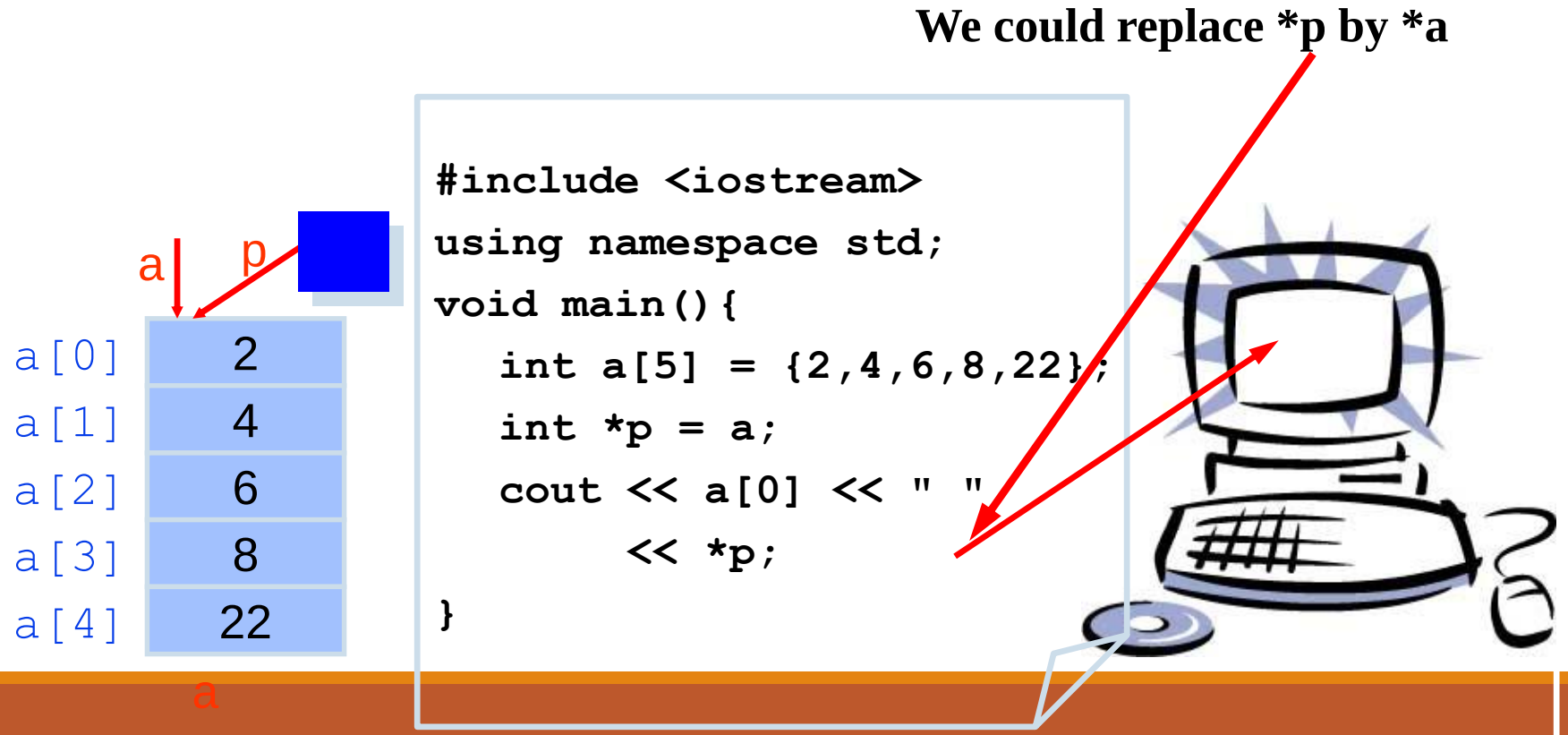
Name as pointer: 0x0065FDE4

Dereferencing An Array Name



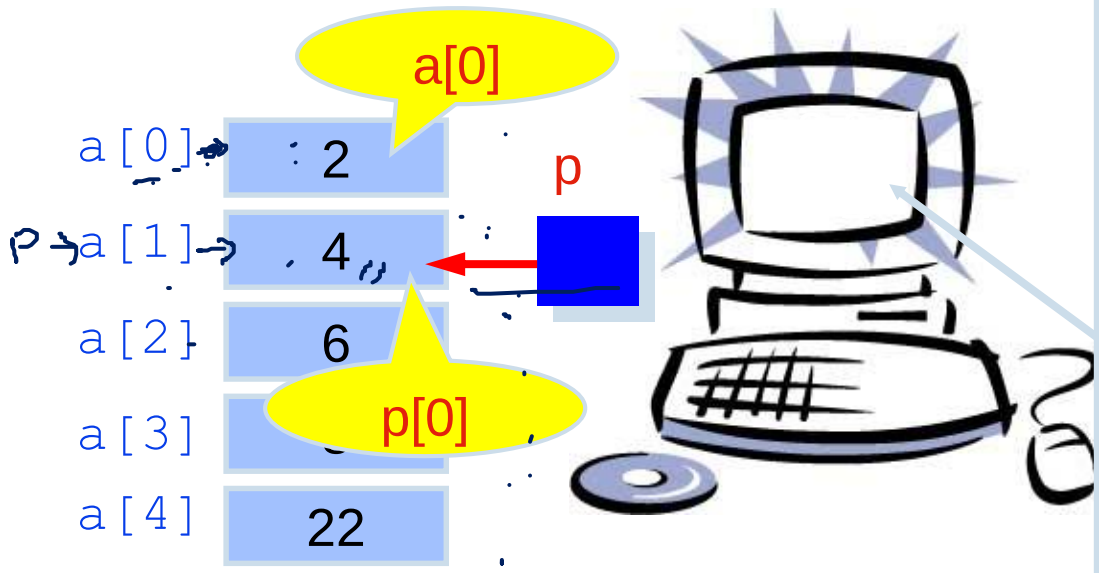
Array Names as Pointers

- To access an array, any pointer to the first element can be used instead of the name of the array.



Multiple Array Pointers

- Both a and p are pointers to the same array.




```
#include <iostream>
using namespace std;
void main(){
   int a[5] = {2,4,6,8,22};
   int *p = &a[1];
   cout << a[0] << " "
        << p[-1];
   cout << a[1] << " "
        << p[0];
}
```

Operations on Pointers I

- Pointer variables can be used in expressions. For example, if **p1** and **p2** are properly declared and initialized pointers, then following are valid:
  - **y= *p1* *p2 ;** same as (*p1) * (*p2)
  - **sum=sum+ *p1;**
  - **z=5 * - *p2 / *p1;** same as (5 * (- (*p2)))/(*p1)
  - **Note:-There is a blank space between / and *; otherwise treated as comment.**
  - ***p2 =*p2 +10;**
  - Addition of integers: **p1 + 2** //address increments by 4 bytes
  - Subtraction of integers from pointers: **p2 - 2** //address decrements by 4 bytes

Operations on Pointers II

- Subtraction of one pointer from another:

  p1 - p2 //difference between two addresses

- Short hand operators may also be used with the pointers.

  p1++;

  - -p2;

  Sum+=*p2;

- Pointers can also be used to compare things using relational operators.

  p1>p2; p1==p2;

  p1!=p2;

Operations on Pointers III

- Invalid statements:
  - Pointers are not used in division and multiplication.
    p1/*p2; p1*p2;
    p1/3; not allowed!

  - Two pointers can not be added.
    p1 + p2 is illegal.

Scale Factor

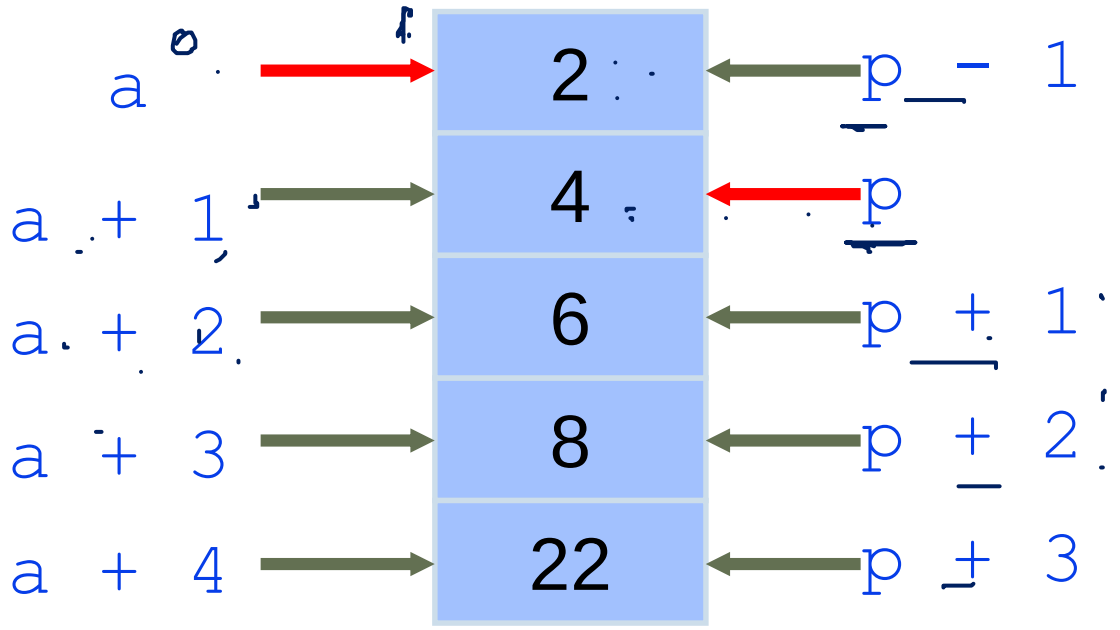
- A scale factor is an increment or decrement given to a pointer to memory address which changes the address of the pointer to next memory address.
- When we increment a pointer , its value is increased by the **length** of the **data type** that it points to. This **length** is called the **scale factor.**
- For example,
  p1++; where p1 is a pointer pointing to some integer variable with an initial value, say **2800**, then after **p1 = p1 + 1**, the value of **p1** will be **2802**.
- We refer to this addition as **pointer arithmetic.**

Valid and Invalid cases

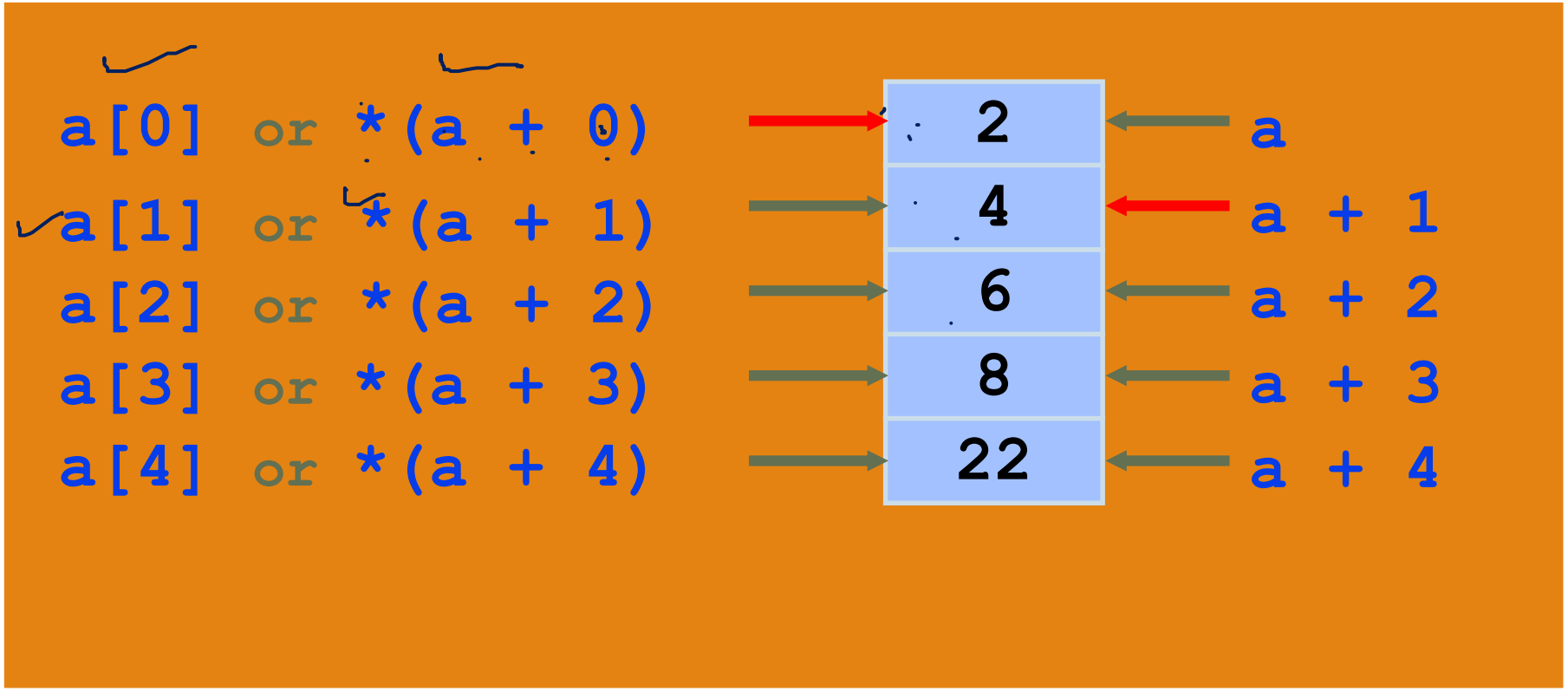
- Invalid Cases:
  - &50 (pointing at constant)
  - int x[10];
    &x (pointing at array name)
  - &(x + y) (pointing at expressions).
- **If x is an array, then expressions such as &x[0] and &x[i+3] are valid and represent the addresses of 0th and (i+3)th elements of x.**

Pointer Arithmetic

- Given a pointer **`p`**, **`p+n`** refers to the element that is offset from **`p`** by **`n`** positions.



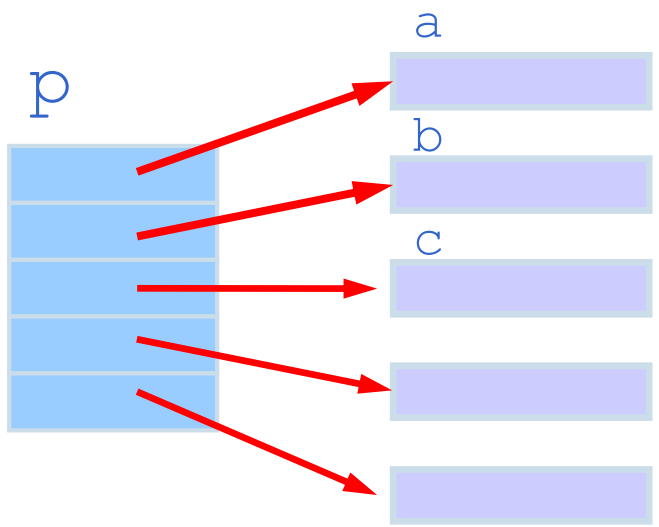
Dereferencing Array Pointers



`*(a+n)` is identical to `a[n]`

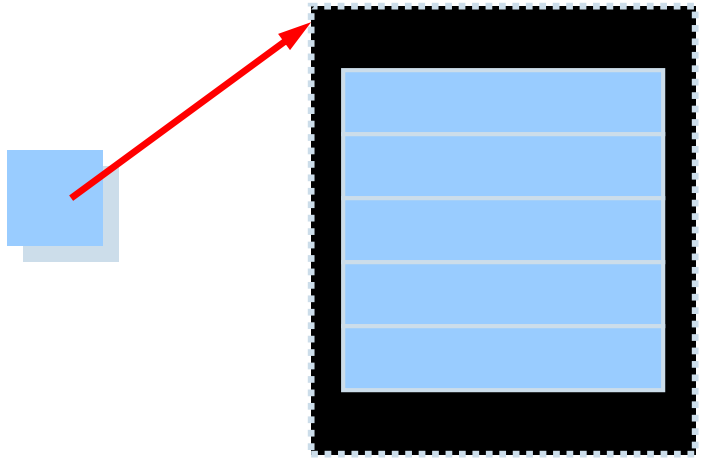
- Note: flexible pointer syntax

Array of Pointers & Pointers to Array



An array of Pointers

```
int a = 1, b = 2, c = 3;
int *p[5];
p[0] = &a;
p[1] = &b;
p[2] = &c;
```



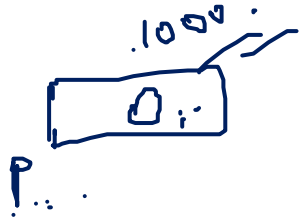
A pointer to an array

```
int list[5] = {9, 8, 7, 6, 5};
int *p;
P = list;//points to 1st entry
P = &list[0];//points to 1st entry
P = &list[1];//points to 2nd entry
P = list + 1; //points to 2nd entry
```

NULL pointer

NULL is a special value that indicates an empty pointer

If you try to access a NULL pointer, you will get an error




```
int *p;
p = 0;
cout << p << endl; //prints 0
cout << &p << endl;//prints address of p
cout << *p << endl;//Error!
```

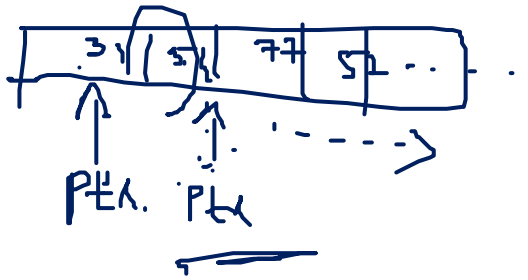
Example

```
#include <iostream.h>
void main( )
{
 int arr[5] = { 31, 54, 77, 52, 93 };
 for(int j=0; j<5; j++) //for each element,
     cout << *(arr+j); //print value
}
```

”arr” itself is a constant pointer which can be used to access the elements.

Example

```
//  array accessed with pointer
#include <iostream.h>
void main()
{
 int arr[] = { 31, 54, 77, 52, 93 }; //array
 int *ptr; //pointer to arr
 ptr = arr; //points to arr
 for(int j=0; j<5; j++) //for each element,
    cout << *(ptr++); //print value
}
```



"ptr" is a pointer which can be used to access the elements.

Problems

- Write a program using pointers to exchange two values (values stored in variables x & y).

- Write a program using pointers to find the sum of all elements stored in an integer array.

Pointers and Character Strings

- The statement **char *cptr =name;** declares cptr as a pointer to a character and assigns address of the first character of name as the initial value
- The statement while(*cptr!='\0') is true until the end of the string is reached.
- When the while loop is terminated, the pointer cptr holds the address of the null character [\0].
- The statement **length = cptr - name;** gives the length of the string name.
- A constant character string always represents a pointer to that string.
- The following statements are valid: **char *name; name ="Delhi";**

Pointers and 2D array I

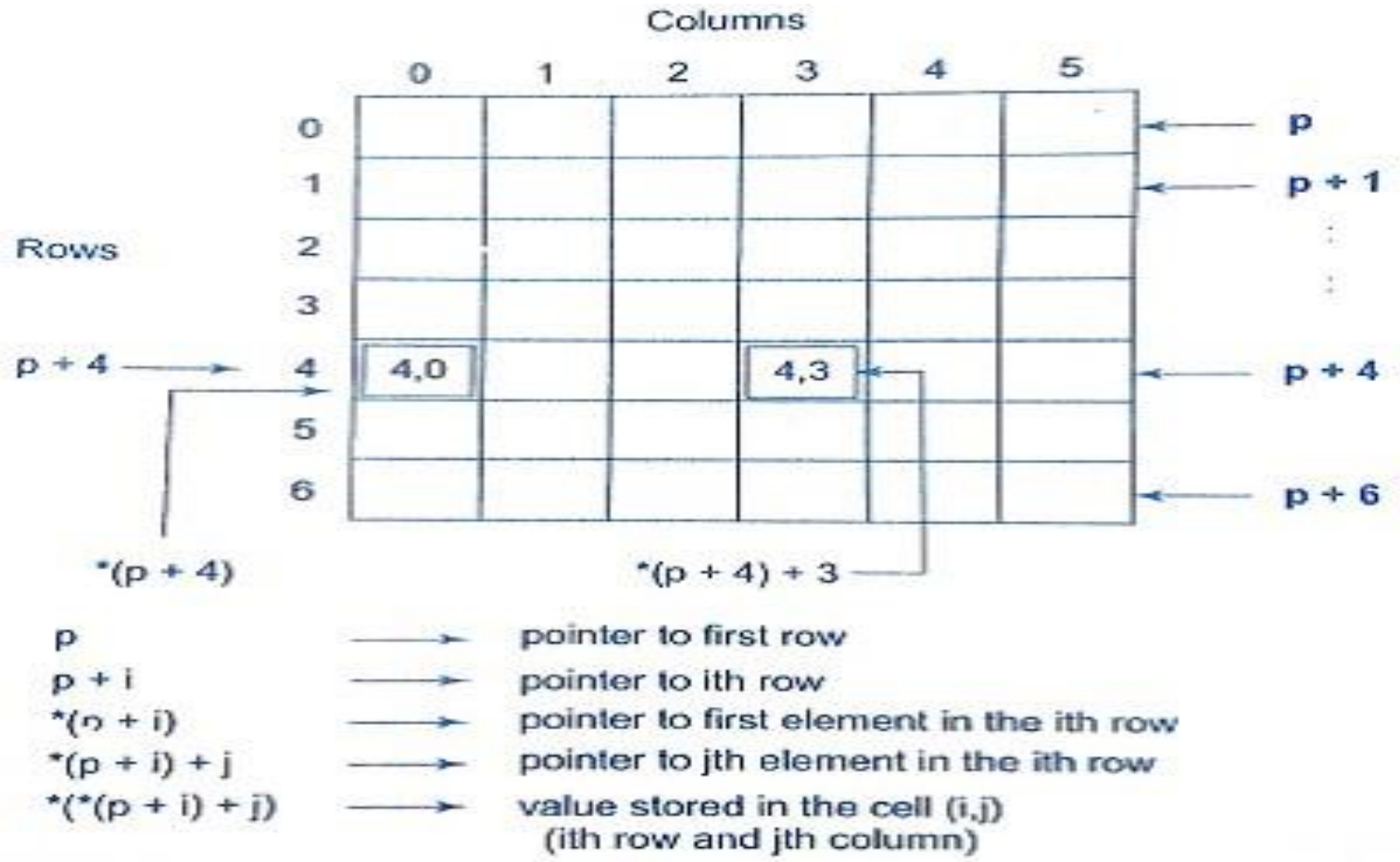
```
int a[][2]={ {12, 22},
             {33, 44} };
int (*p)[2];
p=a;    // initialization
```

- Element in 2d represented as

  ***(*(a+i)+j)** **or**

  ***(*(p+i)+j)**

Pointers and 2D array II



Pointers and 2D array III

```
#include <iostream.h>
void main()
{
 int i,j,(*p)[2], a[][2]={{12, 22}, {33, 44}};
 p=a;
 for(i=0;i<2;i++)
 {
  for(j=0;j<2;j++)
       cout<< *(*(p+i)+j)<<"\t";
  cout<<"\n";
 }}
```

Array of Pointers I

- We can use pointers to handle a table of strings.
  char name[3][25];
  name is a table containing 3 names, each with a maximum length of 25 characters (including '\0')

  Total storage requirement for name is **75 bytes.** But rarely all the individual strings will be equal in lengths.

- We can use a pointer to a string of varying length as
  char *name[3] = { "New Zealand", "Australia", "India" };

Array of Pointers II

- Declares name to be an array of 3 pointers to characters, each pointer pointing to a particular name.

  name[0]→ New Zealand

  name[1]→Australia

  name[2]→India

  This declaration allocates **28 bytes. - Ragged Array**

- The following statement would print out all the 3 names.

```
for(i=0; i<=2;i++)
    cout<<name[i]; or *(name + i);
```

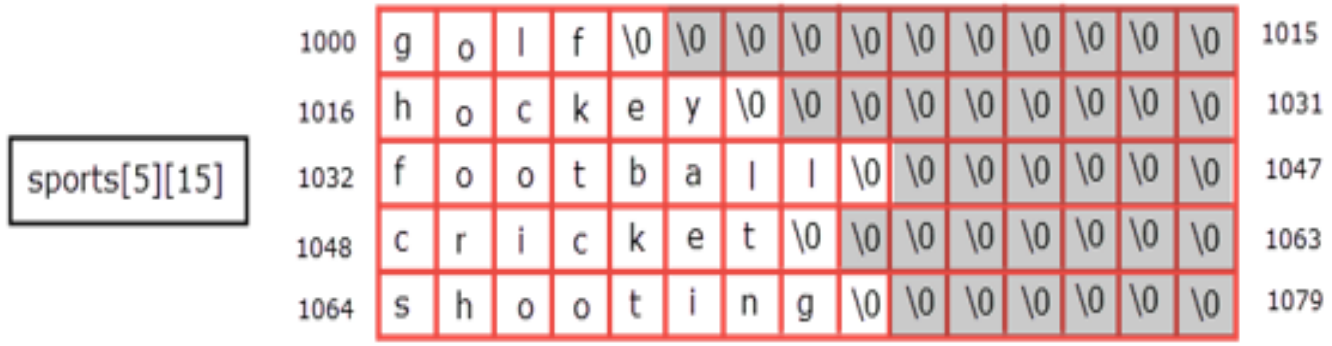
To access the j^{th} character in the i^{th} name, we may write as ***(name[i] + j)**

2D Array for strings

- Not all strings are long enough to fill all the rows of the array, compiler fills these empty spaces with '\0'.
- The total size of the sports array is 75 bytes but only 34 bytes is used, 41 bytes is wasted.
- Therefore, we need a jagged array: A 2-D array whose rows can be of different length.

```
char sports[5][15] = {
                        "golf",
                        "hockey",
                        "football",
                        "cricket",
                        "shooting"
                     };
```

The `sports` array is stored in the memory as follows:



Memory representation of an array of strings or 2-D array of characters

Array of Pointers to Strings

- An array of pointers to strings is an array of character pointers where each pointer points to the first character of the string or the base address of the string.

- To declare and initialize an array of pointers to strings

  char *sports[5] = { "golf", "hockey", "football", "cricket", "shooting“ };

- If initialization of an array is done at the time of declaration then the size of an array can be omitted

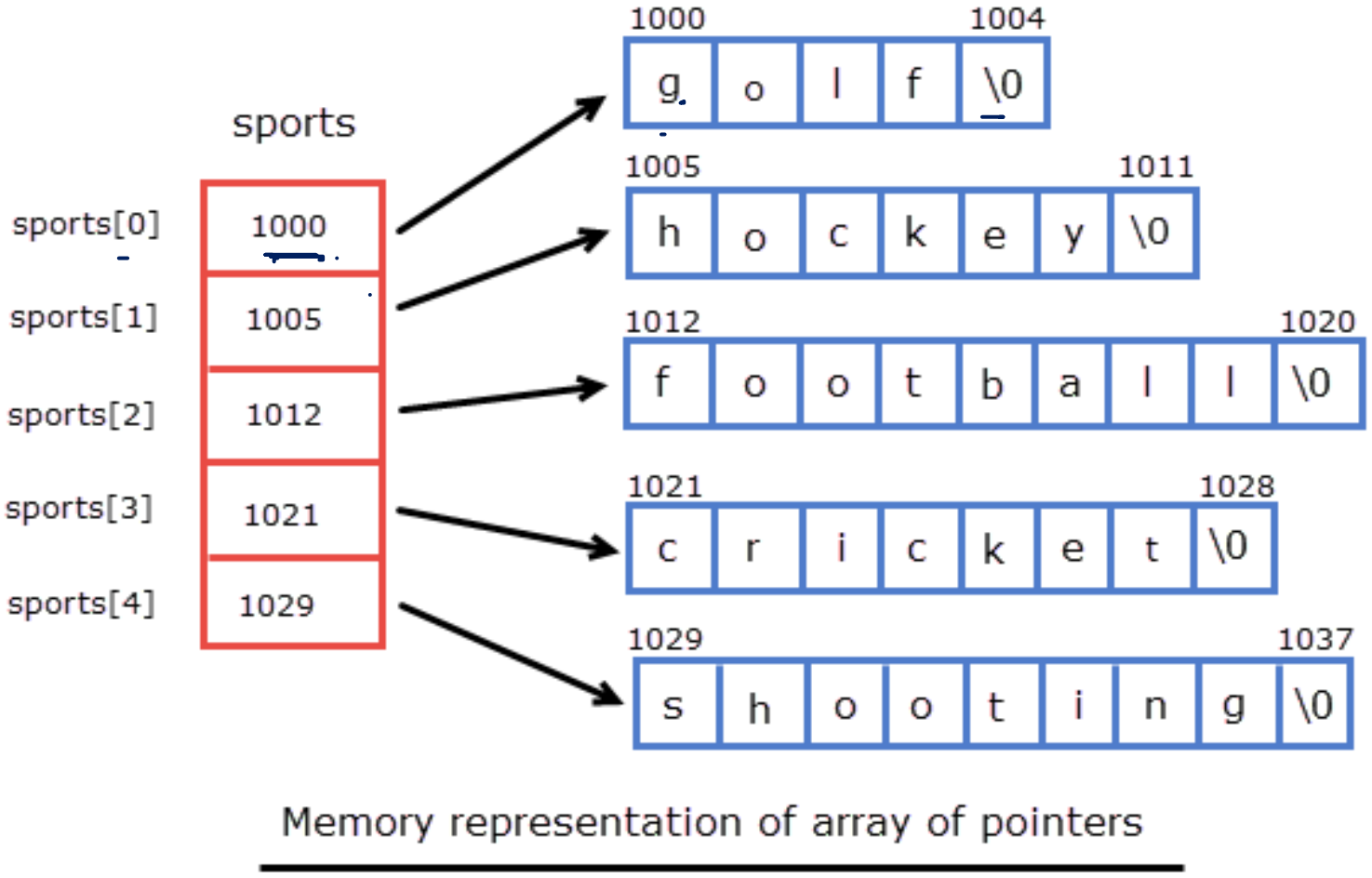
  char *sports[ ] = { "golf", "hockey", "football", "cricket", "shooting“ };

- Space utilized is : 34 + 20 = 54

  In this case, all string literals occupy 34 bytes and 20 bytes are occupied by the array of pointers i.e sports. So, just by creating an array of pointers to string instead of array 2-D array of characters 21 bytes (75-54=21) of memory is saved.

- **The 0th element i.e arr[0] points to the base address of string "golf".**
- **The 1st element i.e arr[1] points to the base address of string "hockey" and so on.**
- An array of pointers to string is stored in memory in this following manner:

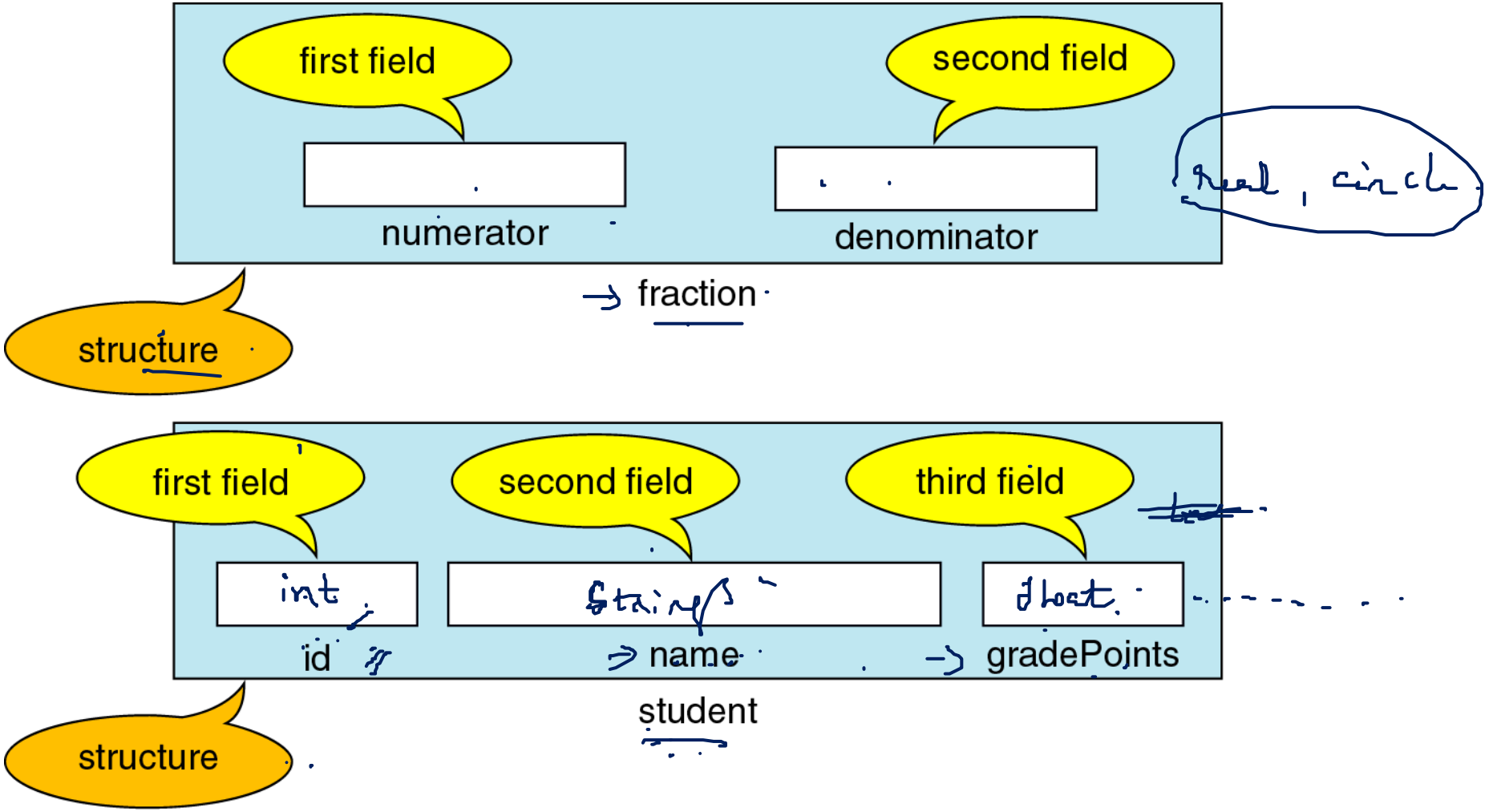
```
char *sports[] = {
                    "golf",
                    "hockey",
                    "football",
                    "cricket",
                    "shooting"
                  };
```



Memory representation of array of pointers

Structures

Figure 12-6 Structure Examples



Note: the data in a structure should all be related to one object

Ex1: both integers belong to the same fraction

Ex2: all data relate to one student

Structure

Structure is a collection of related elements, possibly of different types, having a single name

Field – each element in a structure is called a field

- Same as variable in that it has type and exists in memory
- Differs from a variable in that it is a part of structure

Structure vs. array

- Both are derived data types that can hold multiple pieces of data
- All elements in an array must be of the same type, while the elements in a structure can be of the same or different types

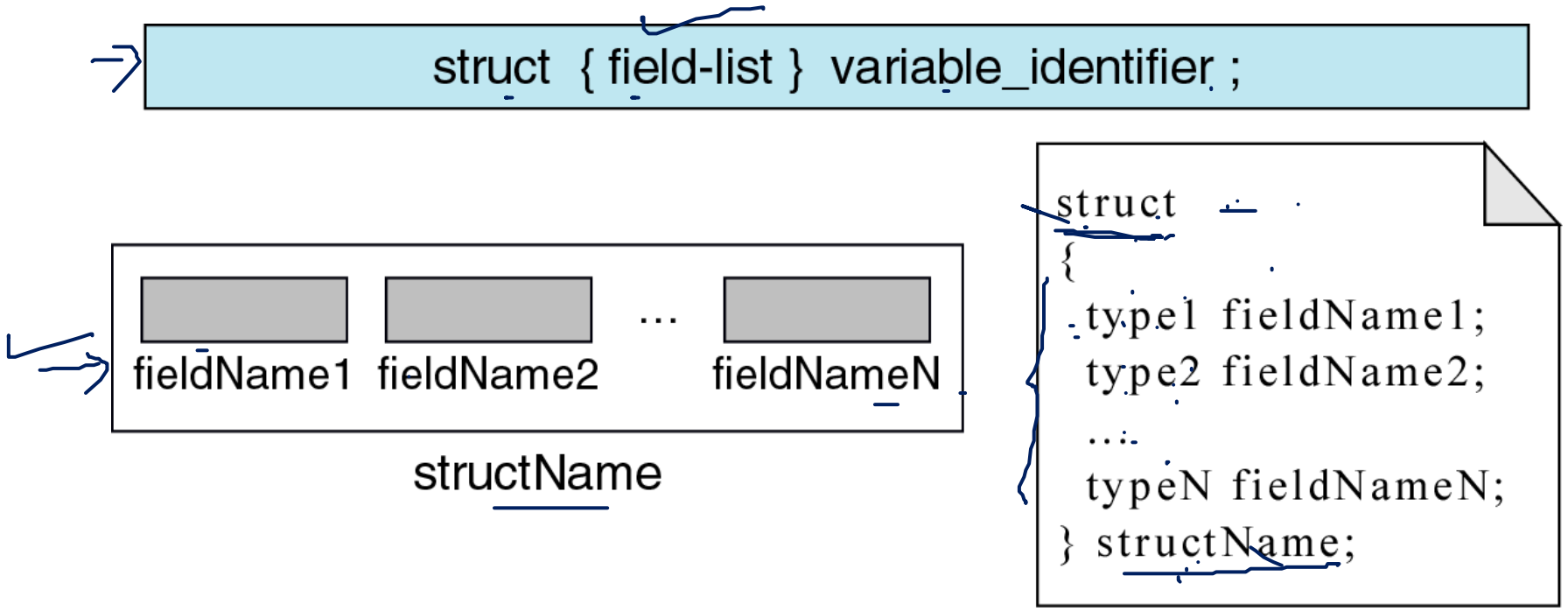
Structure – declaration and definition

Keyword struct – informs the compiler that it is a collection of related data

3 ways to declare/define a structure in C

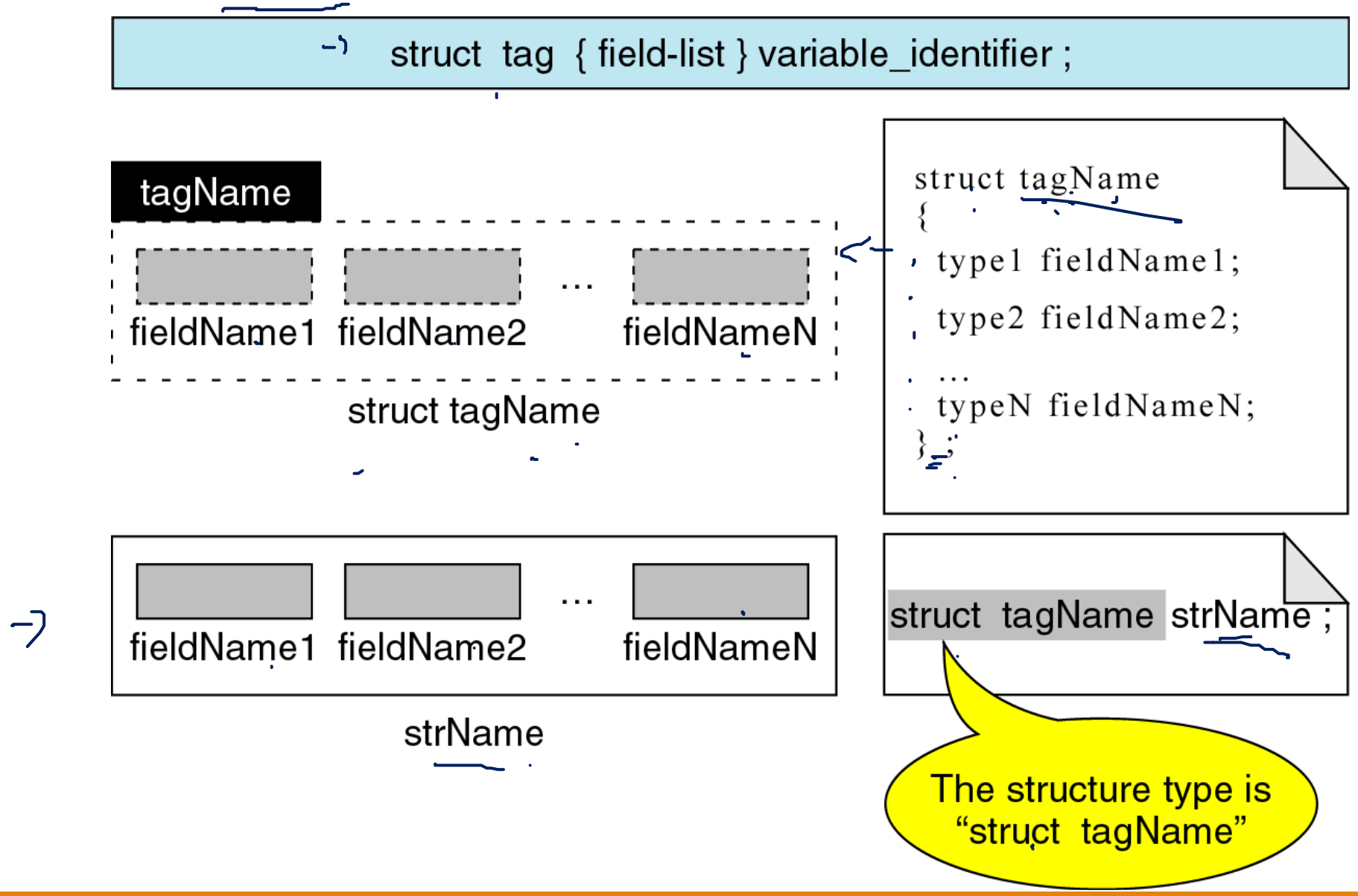
1. Structure variable
2. Tagged structure
3. Type-defined structure

Figure 12-7 structure variable



- **Note: The above definition creates a structure for only one variable definition**
- **As there is no structure identifier (tag), it cannot be shared**
- **So, it is not really a type**
- **This type of declaration format is not to be used**

Figure 12-8 Tagged structure



Tagged Structure – declaration and definition

struct tagname – tagname is the identifier for the structure

- allows to define variables, parameters, and return types

If a struct is concluded with a semicolon after the closing brace, no variables are defined

- So structure is a type template with no associated storage

To define a variable at the same time we declare the structure, list the variables by comma separation

Tagged Structure – declaration and definition

Ex: declare and use student structure:

```
struct student{
        char id[10];
        char name[20];
        int gpa;
};
struct student astudent;
void print_Student(struct student stu);
```

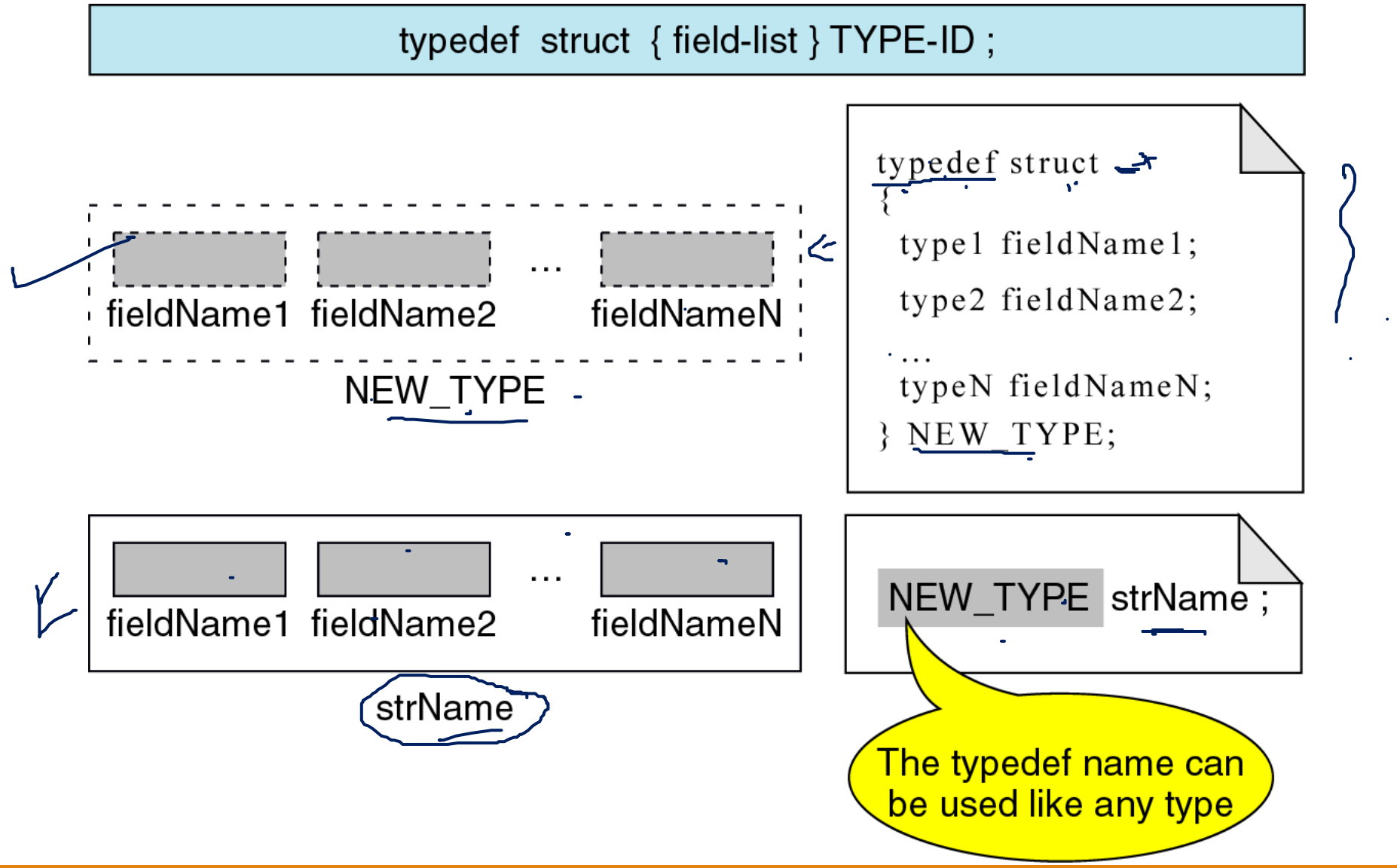
Tagged Structure – declaration and definition

```
struct student{
        char id[10];
        char name[20];
        int gpa;
}; struct student astudent;
void print Student( struct student stu);
```

Follow the above format of declaring structure first and then define variables

- Declare structure declaration in the global area of the program before main
- So structure declaration scope is global and can be shared by all functions

Figure 12-9 Type defined structure



Type defined Structure – declaration and definition

Ex: declare and use student structure:

```
typedef struct {
        char id[10];
        char name[20];
        float gpa;
} STUDENT;
STUDENT astudent;
void printStudent(STUDENT stu);
```

Type defined Structure – declaration and definition

The most powerful way to declare a structure is by **typedef**

Typedef is to be added before struct

- Identifier at the end of the block is the type definition name not a variable
- Cannot define a variable with the typedef declaration

```
typedef struct {
        char id[10];
        char name[20];
        float gpa;
} STUDENT;
STUDENT astudent;
void print Student(STUDENT stu);
```

Type defined Structure – declaration and definition

It is possible to combine the tagged structure and type definition structure in a tagged type definition

The difference between tagged type definition and a type defined structure is

- Here, the structure has a tag name in tagged type definition

```
typedef struct tag {
        char id[10];
        char name[20];
        float gpa;
} STUDENT;
STUDENT astudent;
void printStudent(STUDENT stu);
```

Figure 12-10 struct format variations

```
struct    {
              …
          }  variable_identifier ;
```

structure variable

```
struct  tag
          {
              …
          }  variable_identifier ;

struct  tag  variable_identifier ;
```

tagged structure

```
typedef  struct
          {
              …
          }  TYPE_ID  ;

TYPE_ID   variable_identifier ;
```

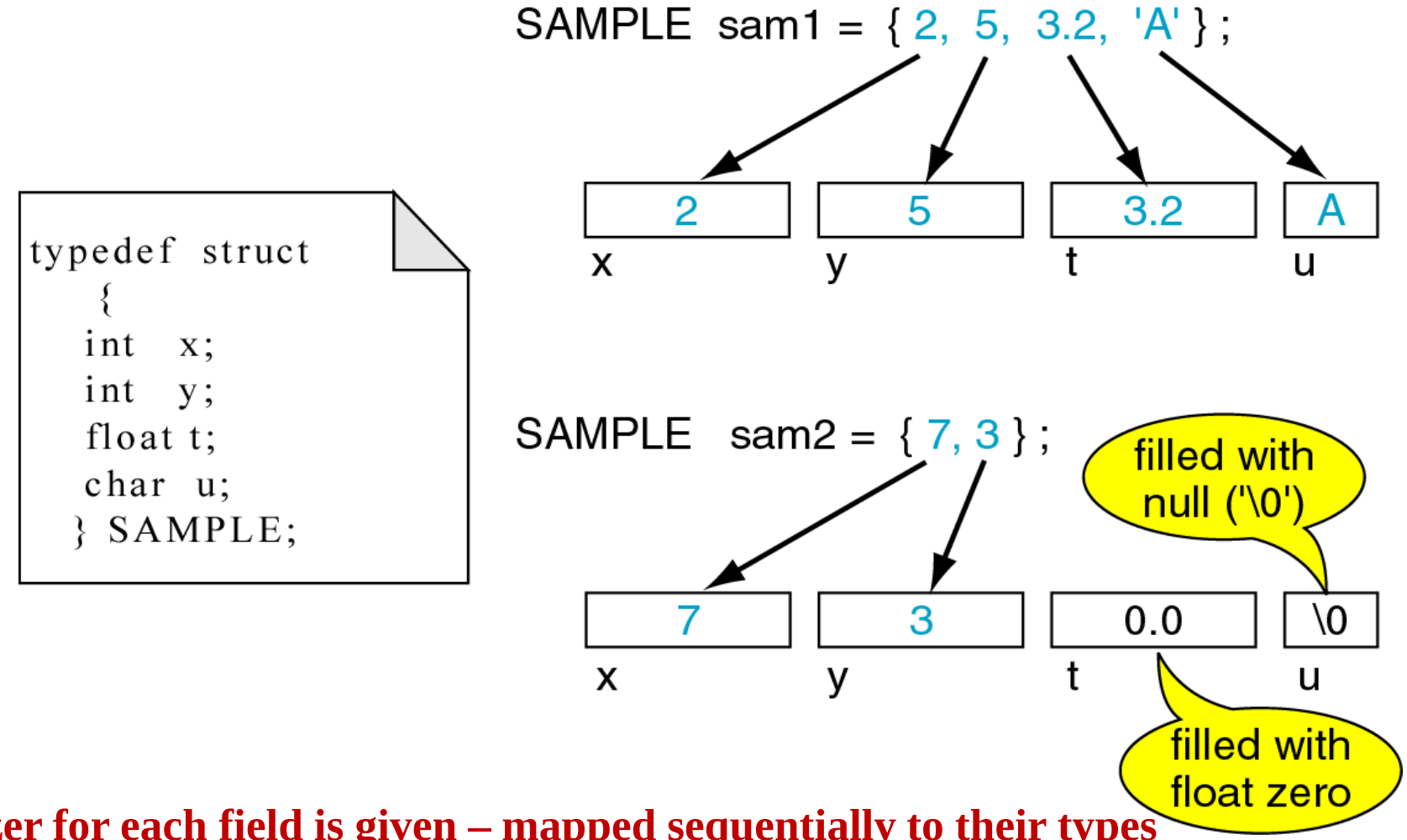
type-defined structure

Initializing structures

Rules for structure initialization are similar to rules of array initialization

- The initializers are enclosed in braces and comma separated
- They must match their corresponding types in the structure definition
- For a nested structure, the nested initializes must be enclosed in their own set of braces

Figure 12-11 Initializing structures



- **Ex1: initializer for each field is given – mapped sequentially to their types**
- **Ex2: initializer for some fields are only given – structure elements will be assigned null values – zero for integers and floating point numbers, \0 for chars and strings (same as in arrays)**

Accessing structures

- Same way as we manipulate variables using expressions and operators, the structure fields can also be operated

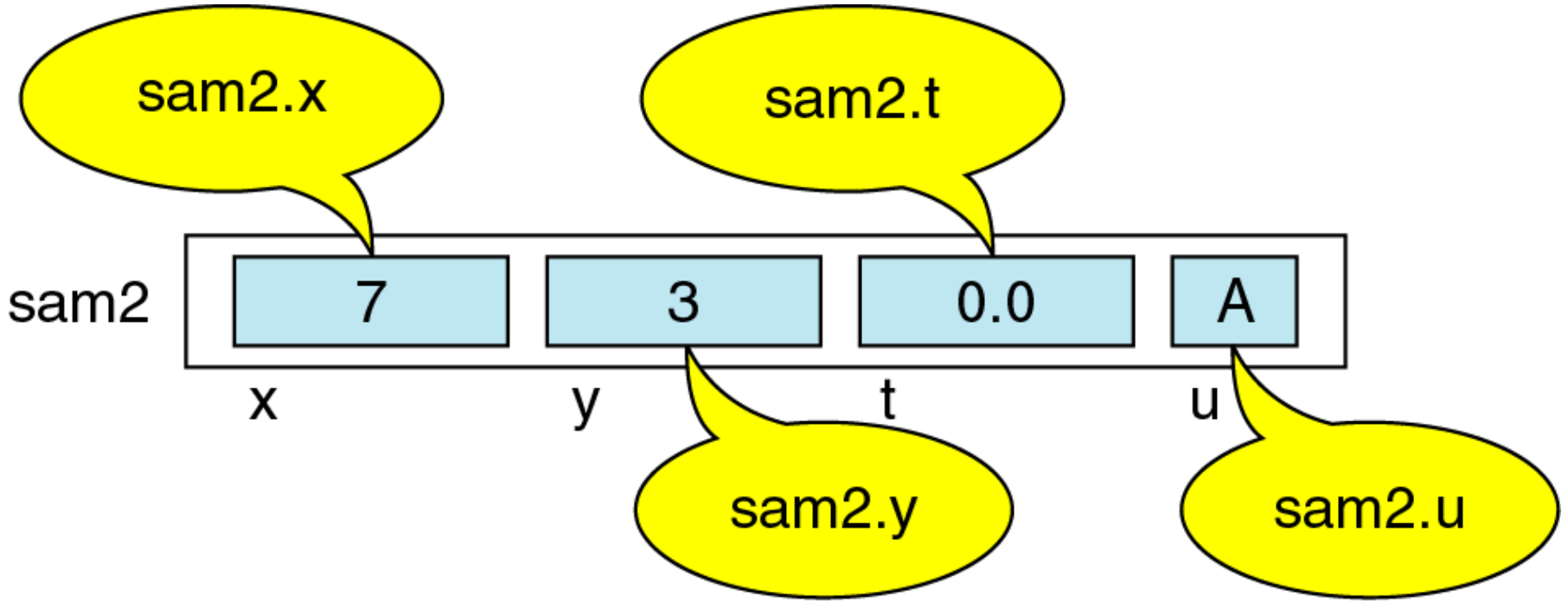
- **Example:**

```
typedef struct {
        char id[10];
        char name[20];
        float gpa;
} STUDENT;
STUDENT astudent;
```

- **Refer to fields by**

astudent.id, astudent.name, astudent.gpa

Figure 12-12 structure member operator



Ex: Reading data into and writing data from structure members is same as done for variables

scanf("%d %d %f %c", &sam2.x, &sam2.y, &sam2.t, &sam2.u);

Precedence of Member operator

Similar to operator [] for array indexing, dot is member operator for structure reference

1) The precedence of member operator is higher than that of increment

Ex: sam2.x++; ++sam2.x;

No parentheses are required

2) **&sam1.x is eqt to &(sam1.x)**. So dot has higher priority than & operator

Operations on Structures

Assignment operation : assigning one structure to another

Structure is treated as one entity and only one operation i.e., assignment is allowed on the structure itself

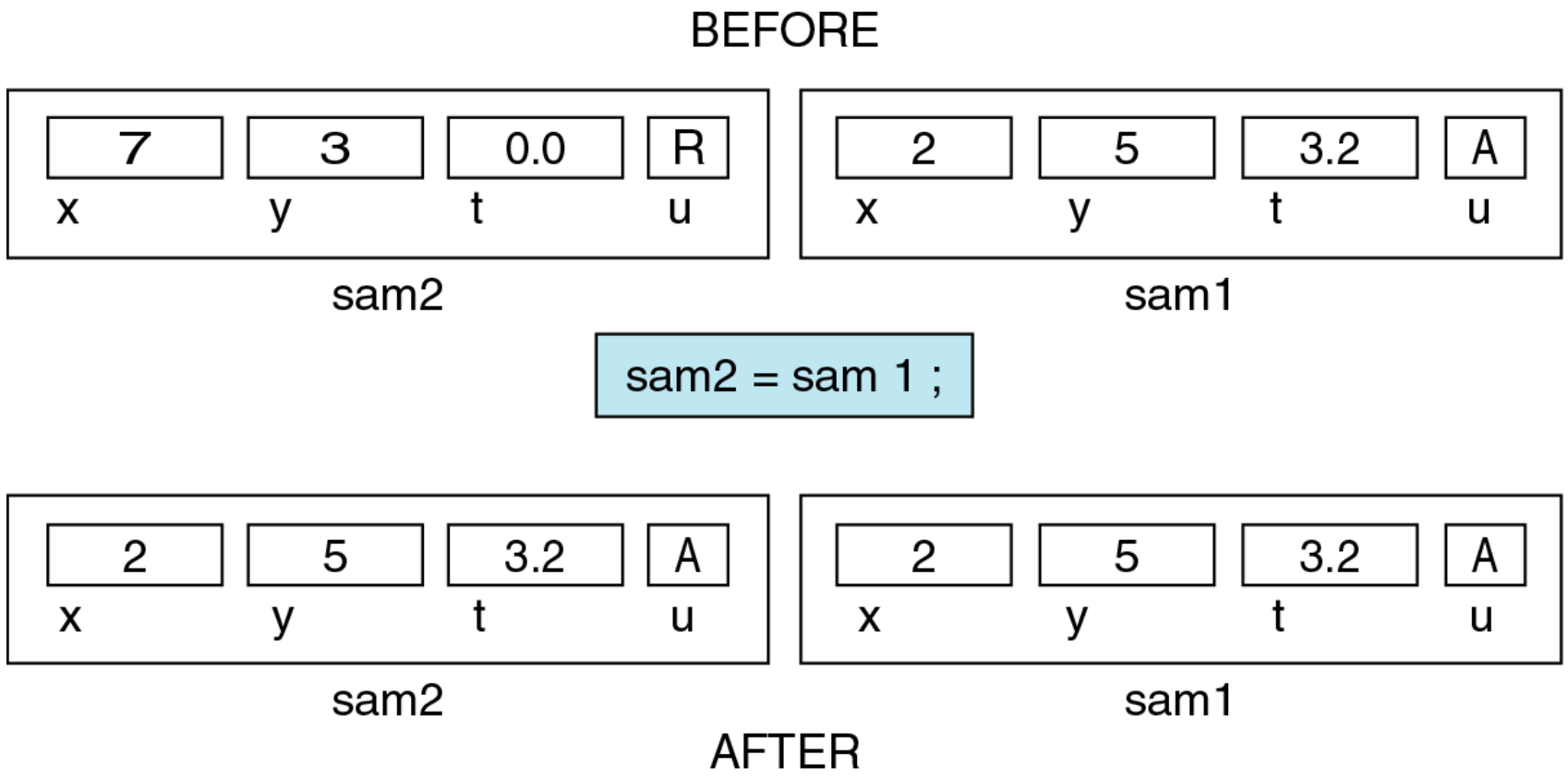
To copy one structure to another structure of the same type

- Rather than assigning individual members
- Assign one to another

Ex: read values into sam1 from the keyboard. Now copy sam1 to sam2 by

sam2=sam1;

Figure 12-13 Copying a structure

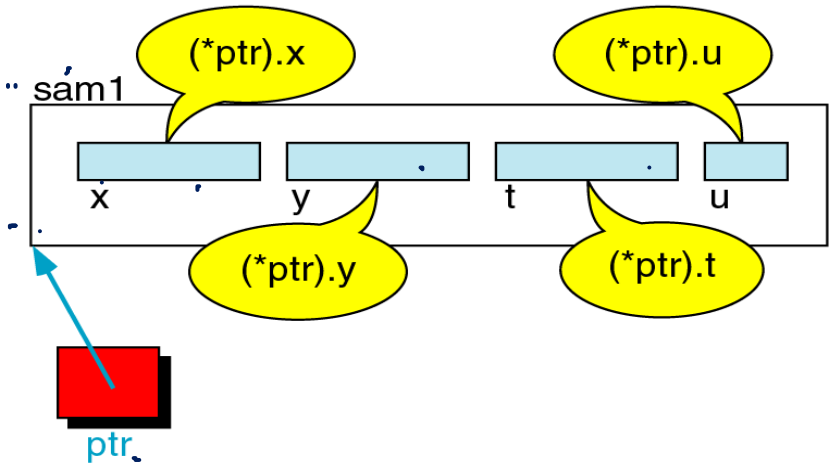


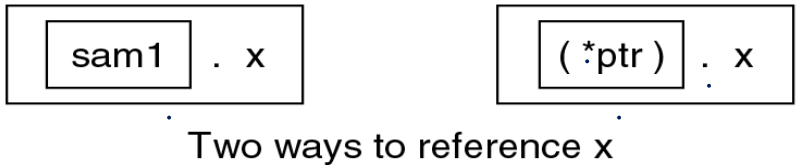
int *p;

Pointer to structures

Like other types, structures can also be accessed through pointers

```
typedef struct
 {
   int   x;
   int   y;
   float t;
   char  u;
  } SAMPLE;
   ...
   SAMPLE   sam1;
   SAMPLE  *ptr;
   ...
   ptr = &sam1;
   ...
```

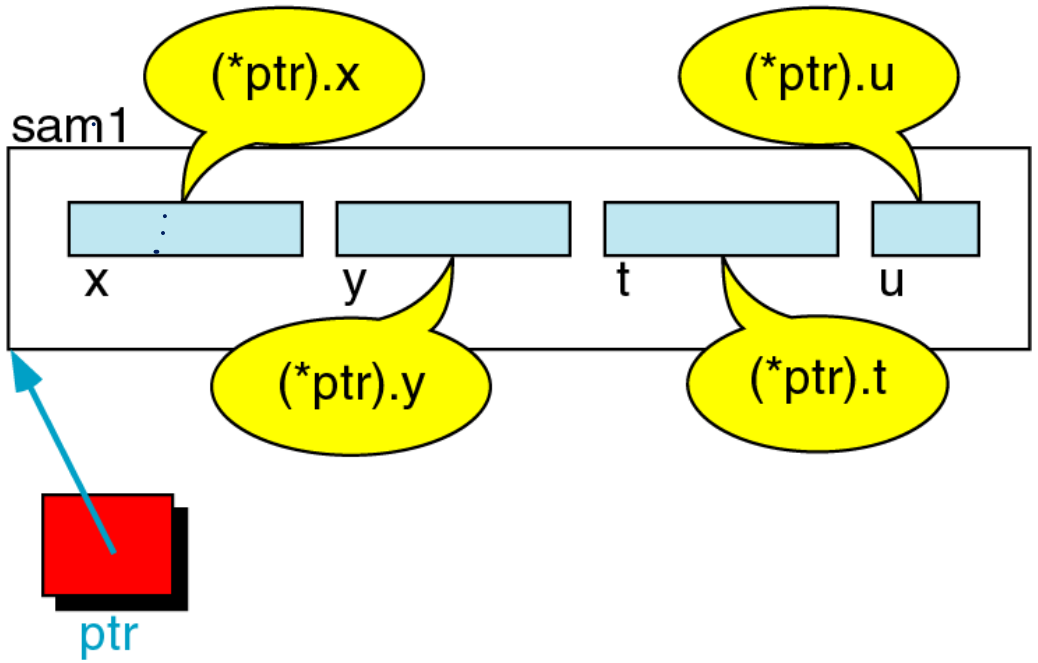


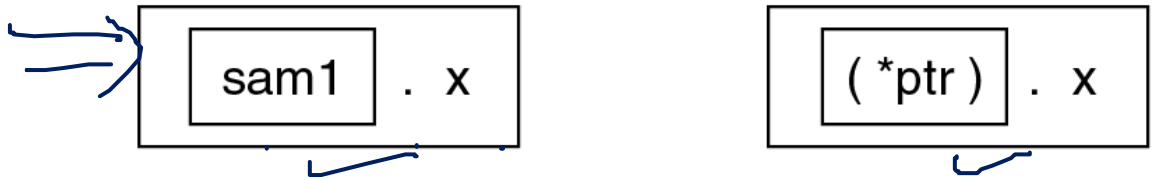


- **Accessing structure itself by *ptr**
- **ptr contains the address of the beginning of the structure**
- **Now we do not only need to use structure name with member operator such as sam1.x**
- **we can also use (*ptr).x**

Figure 12-14 Pointers to structures

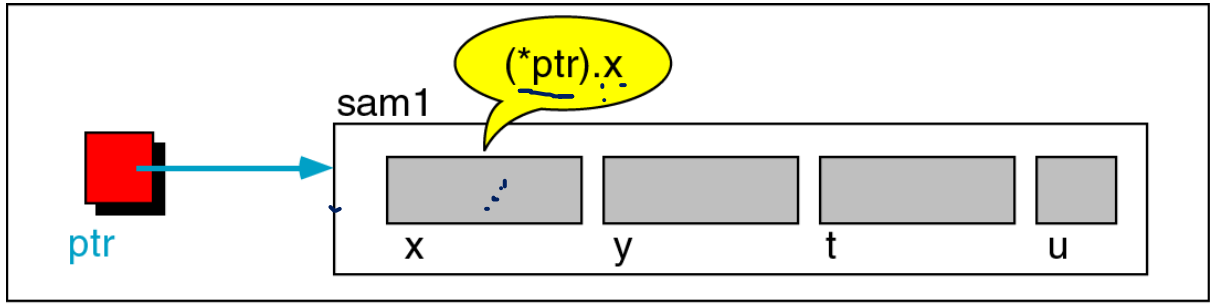
```
typedef struct
  {
   int   x;
   int   y;
   float t;
   char  u;
  } SAMPLE;
   ...
   SAMPLE   sam1;
   SAMPLE  *ptr;
   ...
   ptr = &sam1;
   ...
```



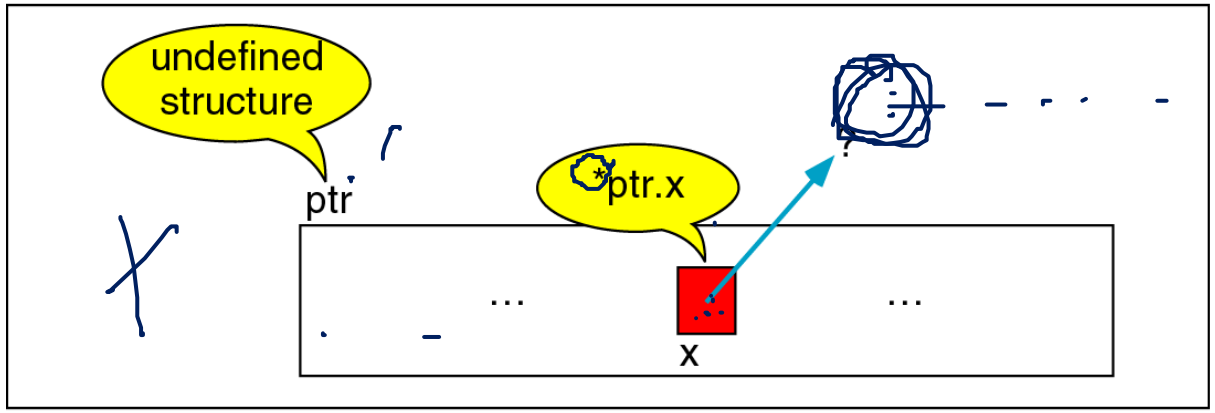


Two ways to reference x

Figure 12-15 Interpretation of invalid pointer use



The correct reference



The wrong way to reference the component

- **In the expression (*ptr).x, parentheses are necessary as member operator has more priority than indirection operator**
- **Default interpretation of *ptr.x is *(ptr.x) which is error because it means that there is a structure called ptr (undefined here) containing a member x which must be a pointer**
- **So, a compile-time error is generated as it is not the case**

Selection operator

However, there is a selection operator **->** (minus sign and greater than symbol) to eliminate the problem of pointer to structures

The priority of selection operator **(->)** and member operator**(.)** are the same

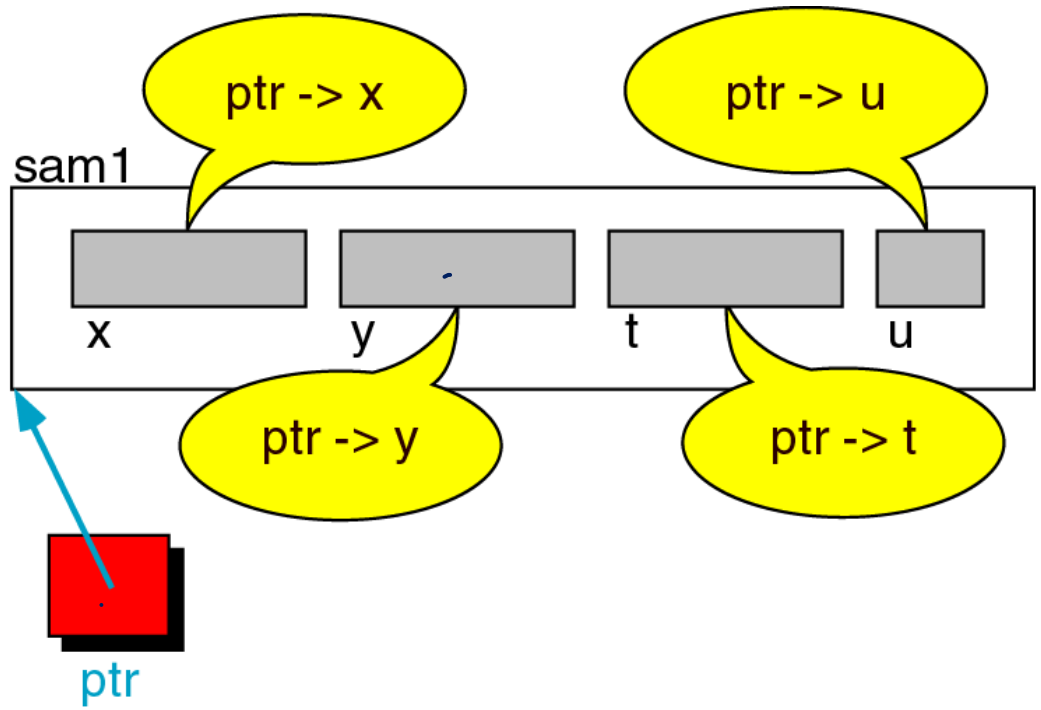
The expressions :

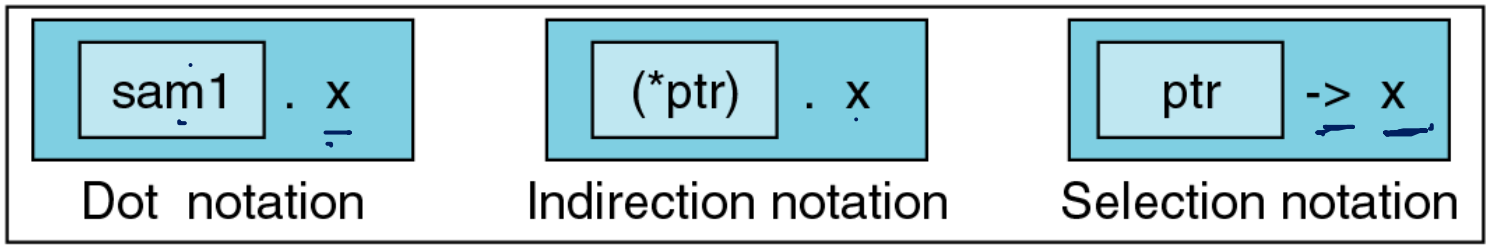
(*pointerName).fieldName is same as pointerName->fieldName

But pointerName -> fieldName is preferred

Figure 12-16 pointer selection operator

```
typedef struct
  {
   int   x;
   int   y;
   float t;
   char  u;
  } SAMPLE;
  ...
  SAMPLE   sam1;
  SAMPLE  *ptr;
  ...
  ptr = &sam1;
  ...
```





three ways to reference the field **x**

Pointers and Structures I

```
struct inventory
{
 char name[30]; int
 number; float price;
}product[2],*ptr;
```

- **ptr=product;**
- Its members are accessed using the following notation

  ptr→name

  ptr→number

  ptr→price

Pointers and Structures II

- The symbol → is called **arrow operator** (also known as member selection operator)
- The data members can also be accessed using

  (*ptr).number

  Parentheses is required because '.' has higher precedence than the operator *

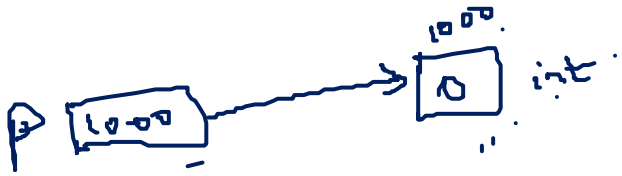
Function returning Pointer

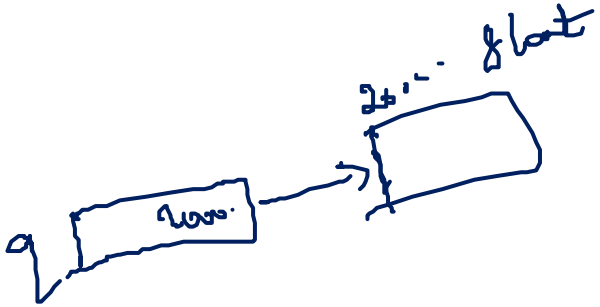
```
int *larger (int *, int  *);
void main()
{
 int a=10, b=20, *p;  p=larger (&a, &b);  cout<<*p;
}
int *larger (int *x, int  *y)
{
 if (*x > *y)
     return(x);  else
     return(y);
}
```

Dynamic Memory Allocation I

- Dynamic memory allocation and deallocation is done using two operators: **new** and **delete**. An object can be created using new operator and destroyed by delete operator.
- A data object created inside a block with new will remain in existence until it is destroyed by using delete.

Pointer_variable = new data_type;





- For example:

  p = new int;

  q = new float;

- Alternatively, **int *p = new int; float *q = new float;**

Dynamic Memory Allocation II

- Initializing memory using new:

ptr variable = new data type(value);

int *p = new int(25);

- The constructor will be called implicitly, to initialize the values which are being passed into the object.
- To create arrays,

ptr variable = new datatype [size];

Dynamic Memory Allocation III

- Creating the multidimensional array all the sizes must be supplied. i.e

```
arr ptr =  new int[3][4][5]; //  Legal
arr ptr =  new int[m][4]; //Legal
arr ptr =  new int[][5];    //  Illegal
arr ptr =  new int[4][];          //  Illegal
```

- The first dimension may be a variable whose value is supplied at runtime, whereas the others must a constant.

DYNAMIC OBJECTS

Memory Management

Static Memory Allocation

- Memory is allocated at compilation time

Dynamic Memory

- Memory is allocated at running time

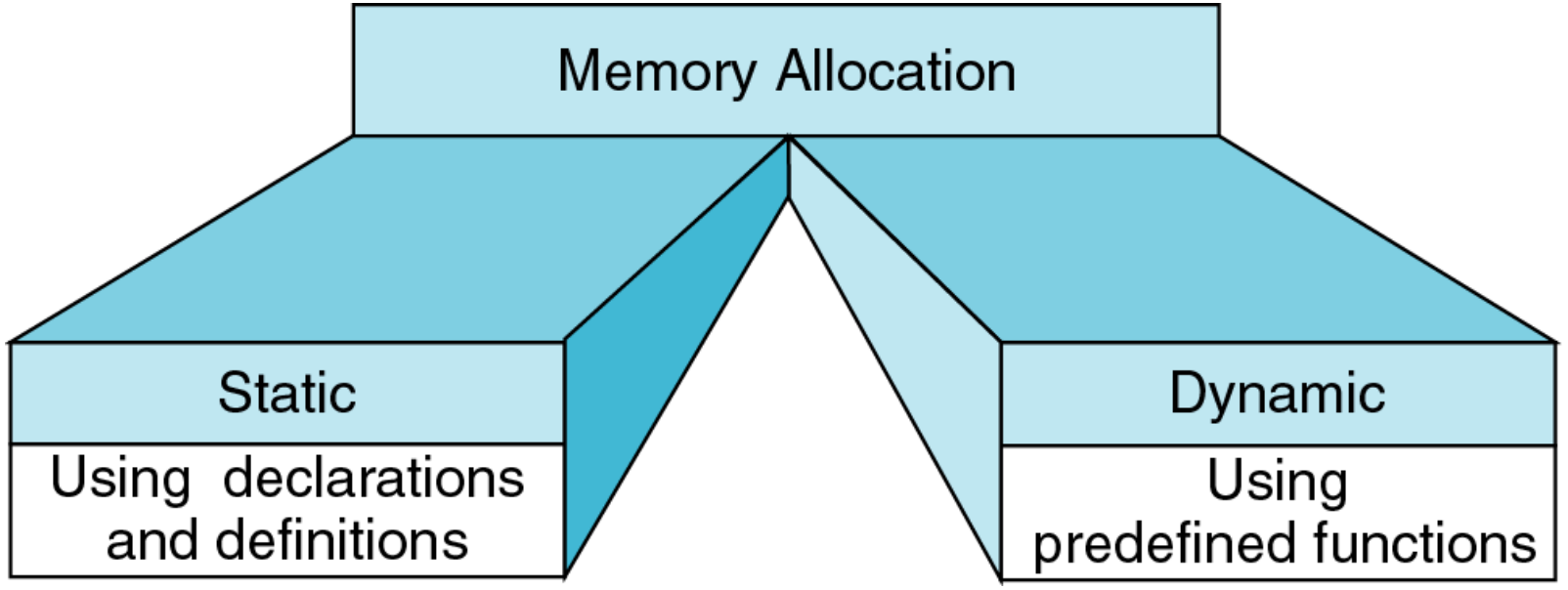
Static vs. Dynamic Objects

- **Static object**

  (variables as declared in function calls)
  - **Memory is acquired automatically**
  - **Memory is returned automatically when object goes out of scope**

- **Dynamic object**
  - **Memory is acquired by program with an allocation request**
    - **new operation**
  - **Dynamic objects can exist beyond the function in which they were allocated**
  - **Object memory is returned by a deallocation request**
    - **delete operation**

Memory Allocation



new
delete

```
{
  int a[200];
  …
}
```

```
int* ptr;
ptr = new int[200];
…
delete [] ptr;
```

Object (variable) creation: New

Syntax

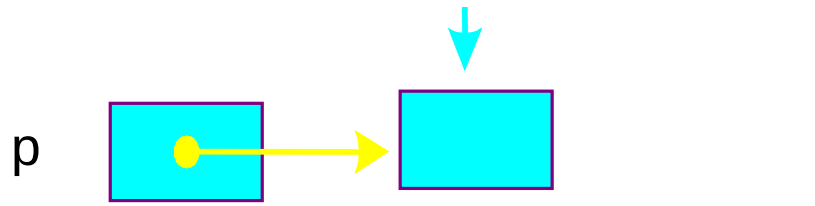
ptr = new SomeType;

where ptr is a pointer of type SomeType

Example:

int* p = new int;

Uninitialized int variable



Object (variable) destruction: Delete

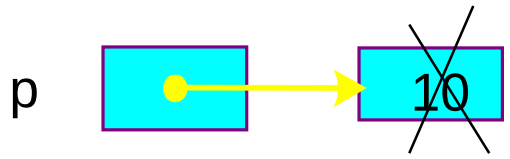
Syntax

delete p;

storage pointed to by p is returned to free store and p is now undefined

Example:

```
int* p = new int;
*p = 10;
delete p;
```



Array of `New`: **`Dynamic arrays`**

Syntax:

P = new SomeType[Expression];

- Where
  - P is a pointer of type SomeType
  - Expression is the number of objects to be constructed -- we are making an array

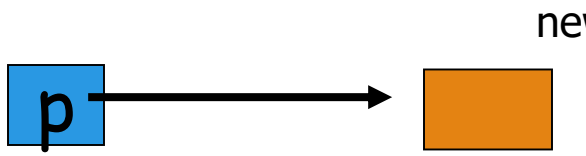
Because of the flexible pointer syntax, P can be considered to be an array

Example

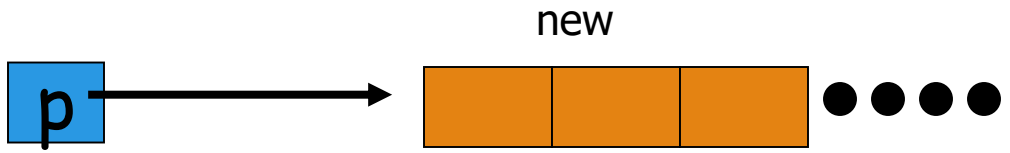
Dynamic Memory Allocation

- **Request for “unnamed” memory from the Operating System**

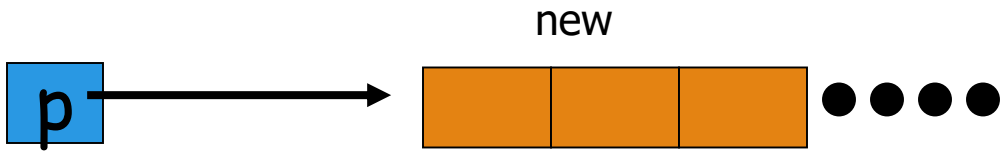
- **int *p, n=10;**
  p = new int;



p = new int[100];



p = new int[n];

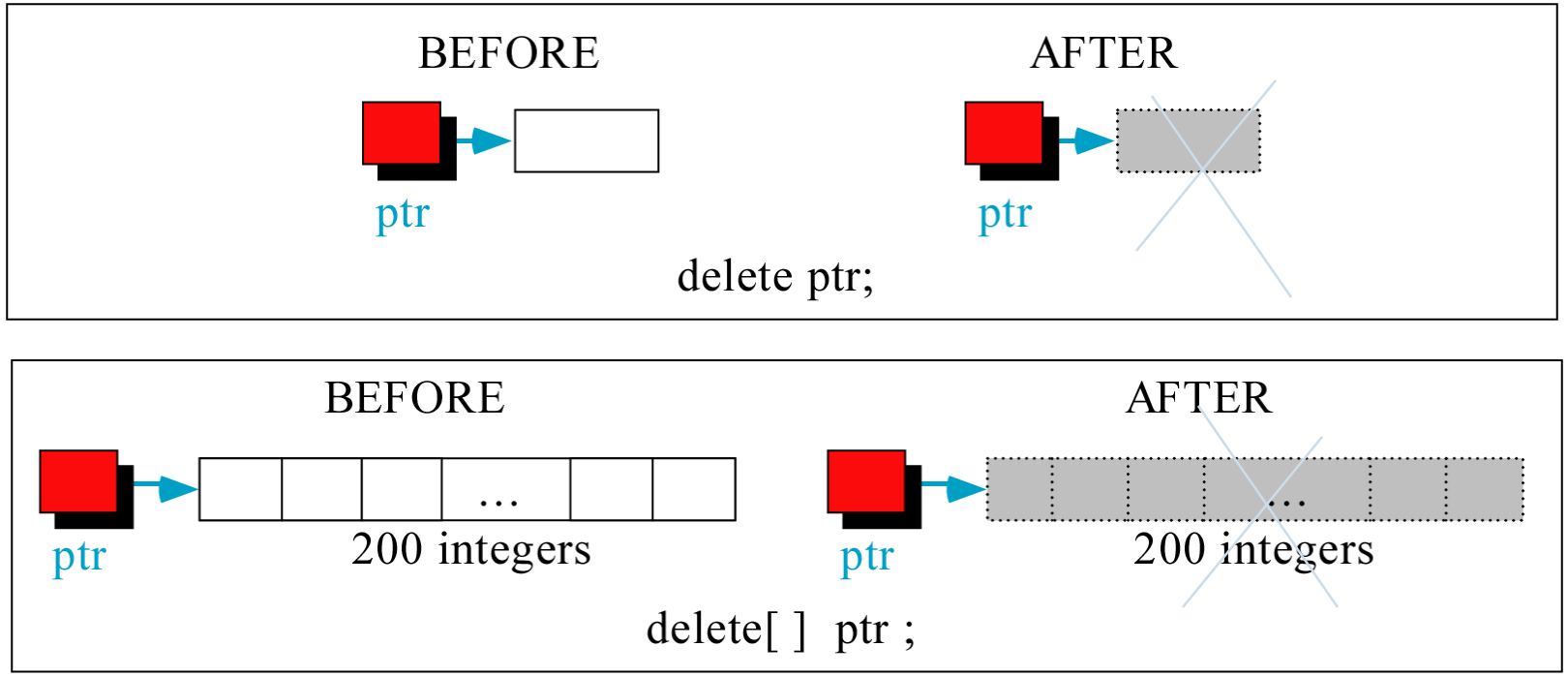


Memory Allocation Example

Want an array of unknown size

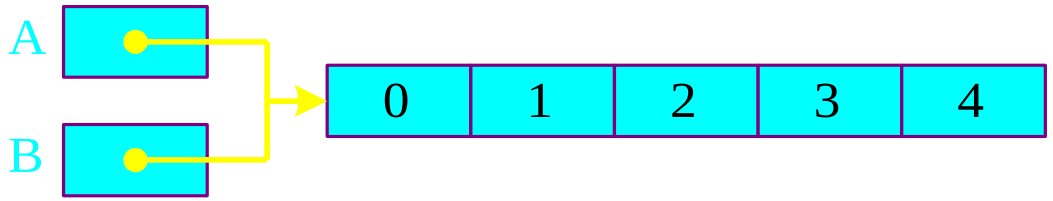
```
#include <iostream>
using namespace std;
void main()
{
      int n;
      cout <<  "How many students? ";
      cin   >> n;
      int *grades = new int[n];
      for(int i=0; i < n; i++){
          int mark;
          cout << "Input Grade for Student" << (i+1)  << " ? :";
          cin >> mark;
          grades[i] = mark;
      }
    . . .
    printMean( grades, n ); // call a function with dynamic array
    . . .
}
```

Freeing (or deleting) Memory

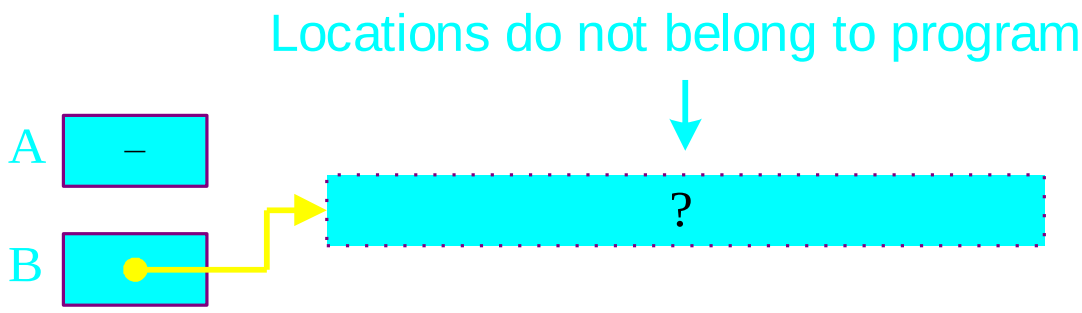


Dangling Pointer Problem

```
int *A = new int[5];
for(int i=0; i<5; i++)
      A[i] = i;
int *B = A;
```

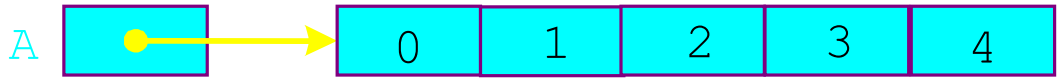



```
delete [] A;
B[0] = 1; // illegal!
```



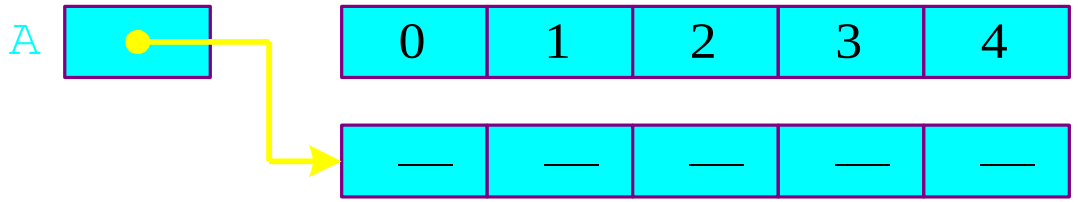
Memory Leak Problem

```
int *A = new int [5];
for(int i=0; i<5; i++)
    A[i] = i;
```



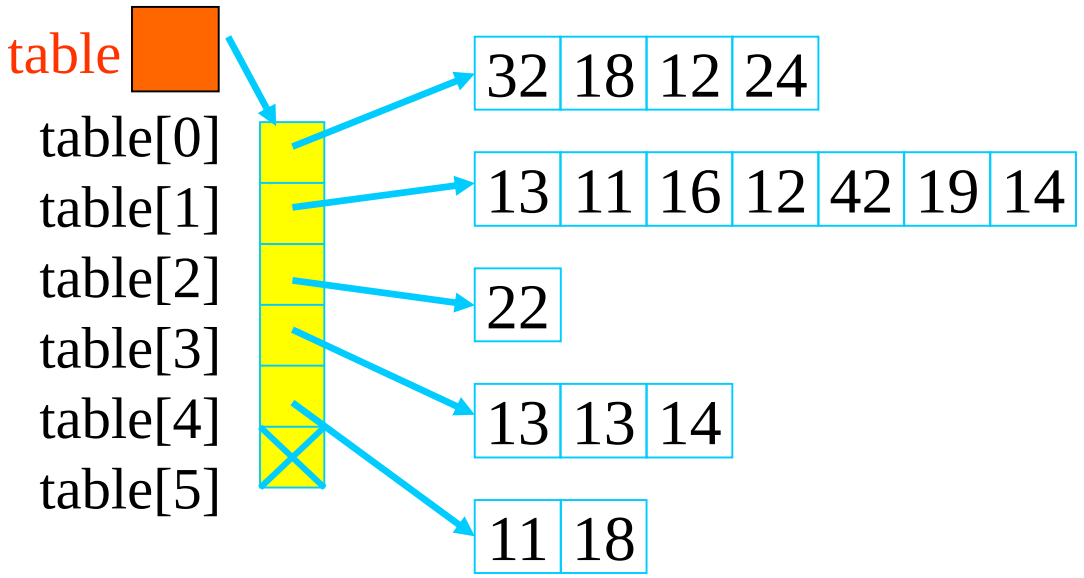

```
A = new int [5];
```

These locations cannot be accessed by program



A Dynamic 2D Array

- **A dynamic array is an array of pointers to save space when not all rows of the array are full.**
- **int **table;**




```
table = new int*[6];
…
table[0] = new int[4];
table[1] = new int[7];
table[2] = new int[1];
table[3] = new int[3];
table[4] = new int[2];
table[5] = NULL;
```

Memory Allocation

```
int **table;
table = new int*[6];

table[0]= new int[3];
table[1]= new int[1];
table[2]= new int[5];
table[3]= new int[10];
table[4]= new int[2];
table[5]= new int[6];
```

```
table[0][0] = 1; table[0][1] = 2;
table[0][2] = 3;

table[1][0] = 4;

table[2][0] = 5; table[2][1] = 6;
table[2][2] = 7; table[2][3] = 8;
 table[2][4] = 9;

table[4][0] = 10; table[4][1] = 11;

cout << table[2][5] << endl;
```

Memory Deallocation

Memory leak is a serious bug!

Each row must be deleted individually

Be careful to delete each row before deleting the table pointer.

- **for(int i=0; i<6; i++)**
  delete [ ] table[i];

delete [ ] table;

Create a matrix of any dimensions, m by n:

```
int m, n;
cin >> m >> n >> endl;

int** mat;

mat = new int*[m];

for (int i=0;i<m;i++)
  mat[i] = new int[n];
```

Put it into a function:

```
int m, n;
cin >> m >> n >> endl;
int** mat;
mat = imatrix(m,n);
…
int** imatrix(nr, nc) {
  int** m;
  m = new int*[nr];
  for (int i=0;i<nr;i++)
     m[i] = new int[nc];
  return m;}
```

Pointers to objects

Pointers to objects

Any type that can be used to declare a variable/object can also have a pointer type.

Consider the following class:

```
class Rational
{
  private:
    int numerator;
    int denominator;
  public:
    Rational(int n, int d);
    void Display();
};
```

Pointers to objects (Cont..)

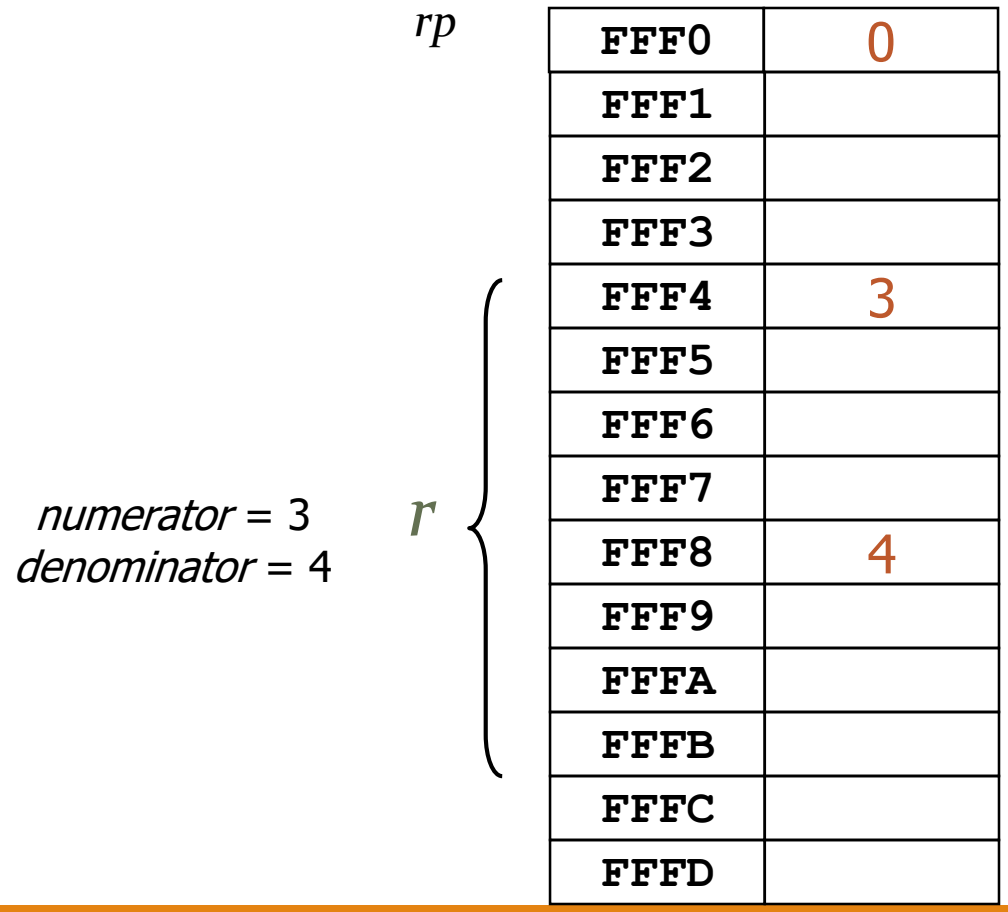
Rational *rp = NULL;
Rational r(3,4);
rp = &r;

rp

FFF0	0
FFF1	
FFF2	
FFF3	
FFF4	
FFF5	
FFF6	
FFF7	
FFF8	
FFF9	
FFFA	
FFFB	
FFFC	
FFFD	

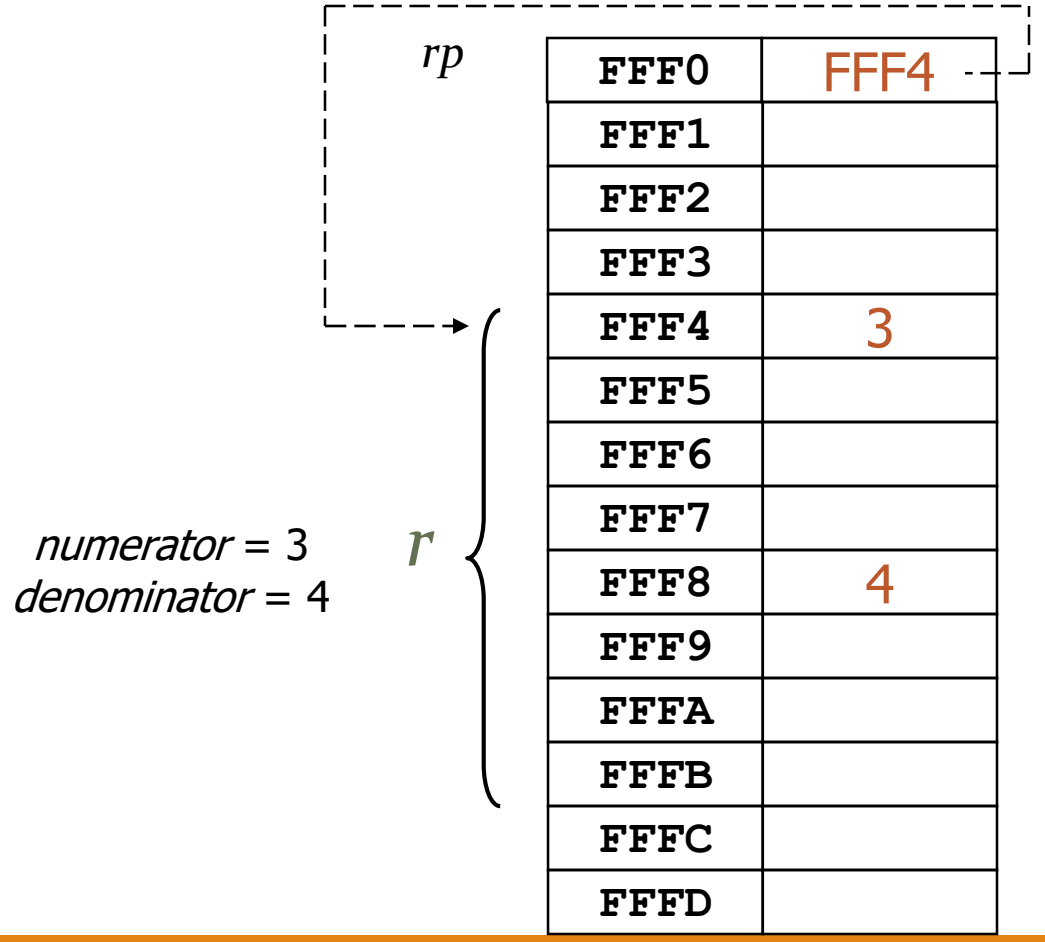
Pointers to objects (Cont..)

```
Rational *rp = NULL;
Rational r(3,4);
rp = &r;
```



Pointers to objects (Cont..)

Rational *rp = NULL;
Rational r(3,4);
rp = &r;



Pointers to objects (Cont..)

If **rp** is a pointer to an object, then two notations can be used to reference the instance/object **rp** points to.

Using the *de-referencing* operator ***

```
(*rp).Display();
```

Using the *member access* operator **->**

```
rp -> Display();
```

Dynamic Allocation of a Class Object

Consider the Rational class defined before:

```
Rational *rp;
int a, b;
cin >> a >> b;
rp = new Rational(a,b);
(*rp).Display(); // rp->Display();
delete rp;
rp = NULL;
```

End of Pointers


LISTS.
General Linear Lists

Objectives.

- Explain the design, use, and operation of a linear list
- Implement a linear list using a linked list structure
- Understand the operation of the linear list ADT
- Write application programs using the linear list ADT
- Design and implement different link-list structures

Array versus Linked Lists

- Arrays are suitable for:
  - Inserting/deleting an element at the end.
  - Randomly accessing any element.
  - Searching the list for a particular value.
- Linked lists are suitable for:
  - Inserting an element.
  - Deleting an element.
  - Applications where sequential access is required.
  - In situations where the number of elements cannot be predicted beforehand.

List is an Abstract Data Type

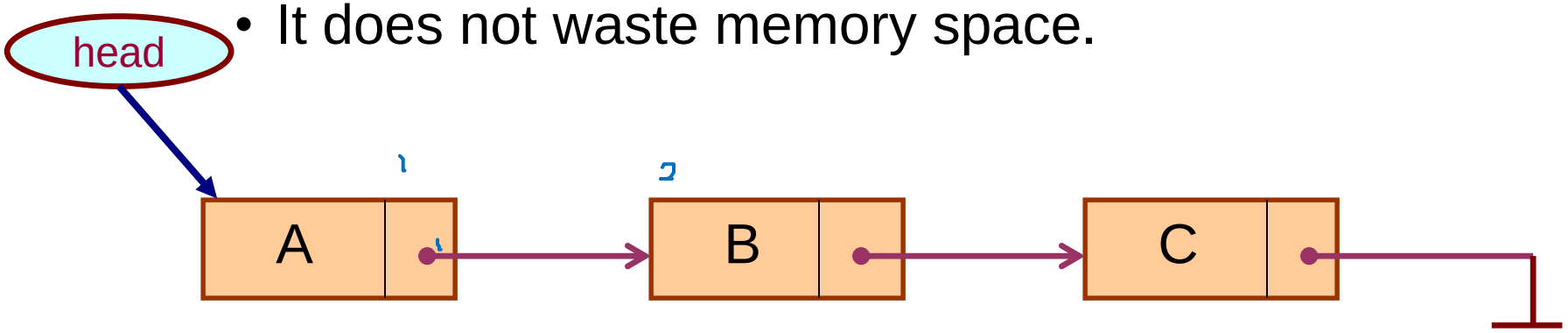
- What is an abstract data type?
  - It is a data type defined by the user.
  - Typically more complex than simple data types like *int*, *float*, etc.
- Why abstract?
  - Because details of the implementation are hidden.
  - When you do some operation on the list, say insert an element, you just call a function.
  - Details of how the list is implemented or how the insert function is written is no longer required.

General linear lists.

- A general linear list is a list in which operations can be done anywhere in the list.
- For simplicity, we refer to general linear lists as lists.

Introduction

- A linked list is a data structure which can change during execution.
  - Successive elements are connected by pointers.
  - Last element points to `NULL`.
  - It can grow or shrink in size during execution of a program.
  - It can be made just as long as required.
  - It does not waste memory space.



- Keeping track of a linked list:
  - Must know the pointer to the first element of the list (called *start*, *head*, etc.).
- Linked lists provide flexibility in allowing the items to be rearranged efficiently.
  - Insert an element.
  - Delete an element.

Basic Operations.

- Insertion
- Deletion
- Retrieval
- Traversal

Insertion.

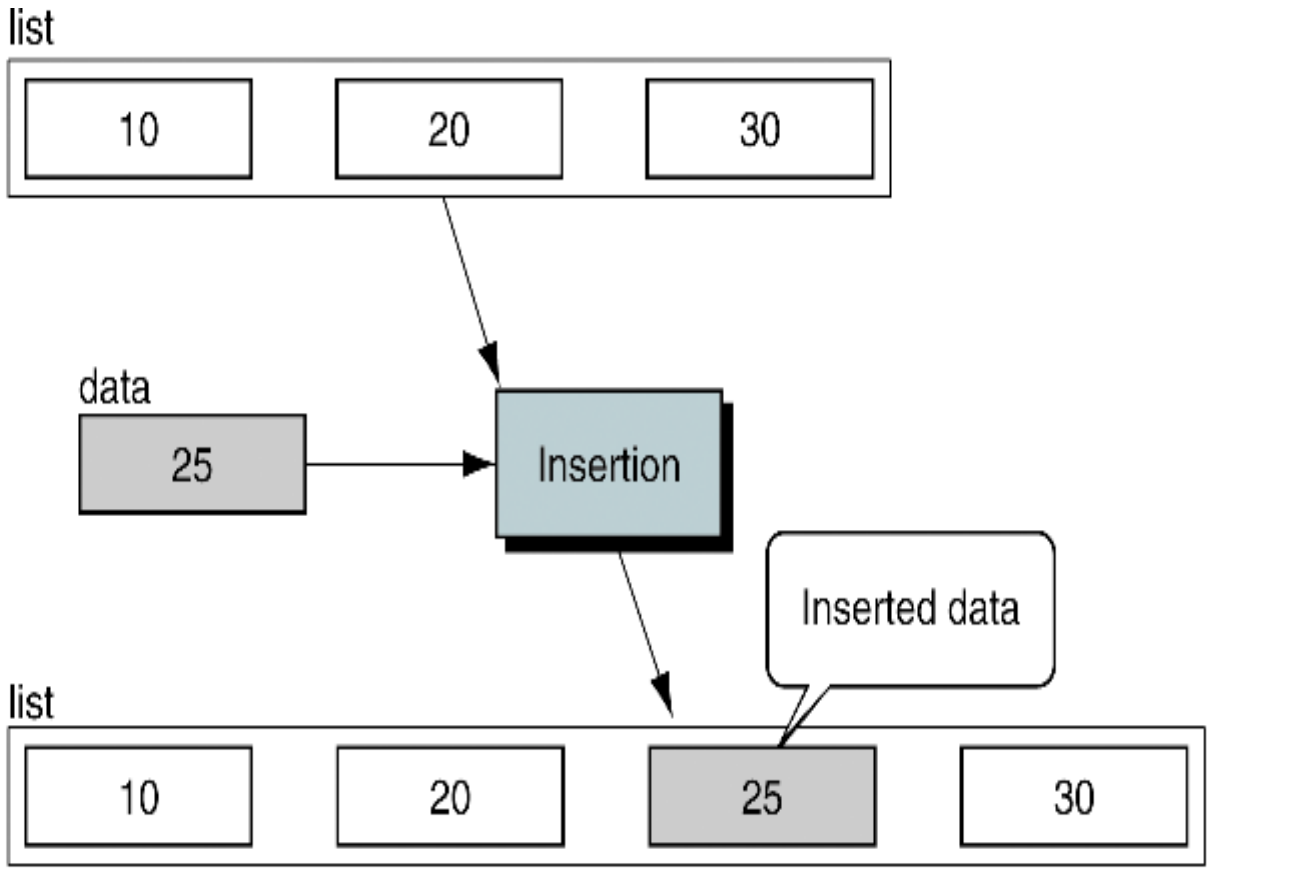


FIGURE 5-1 Insertion

Insertion is used to add a new element to the list

Deletion.

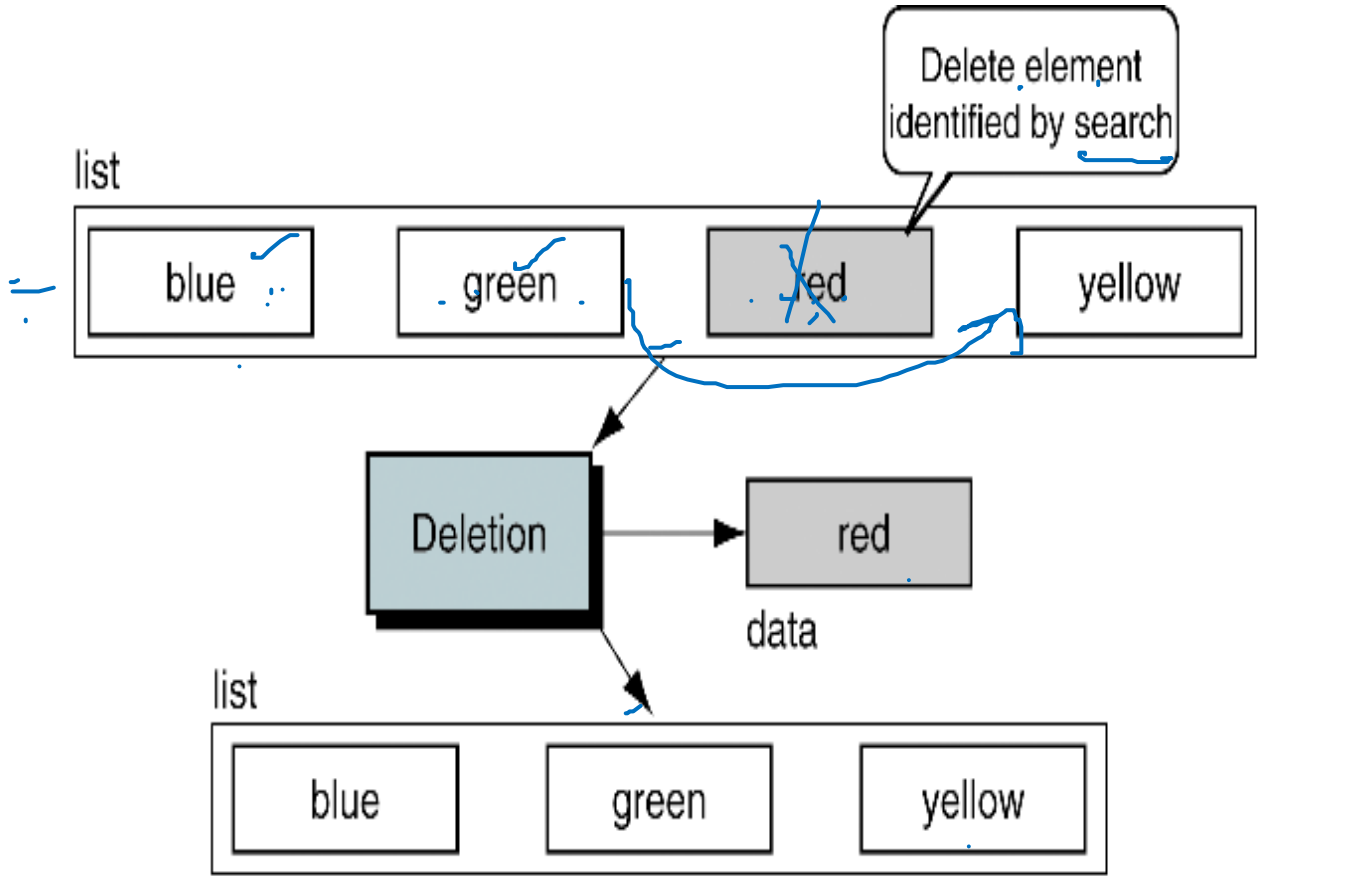


FIGURE 5-2 Deletion

Deletion is used to remove an element from the list.

Retrieval.

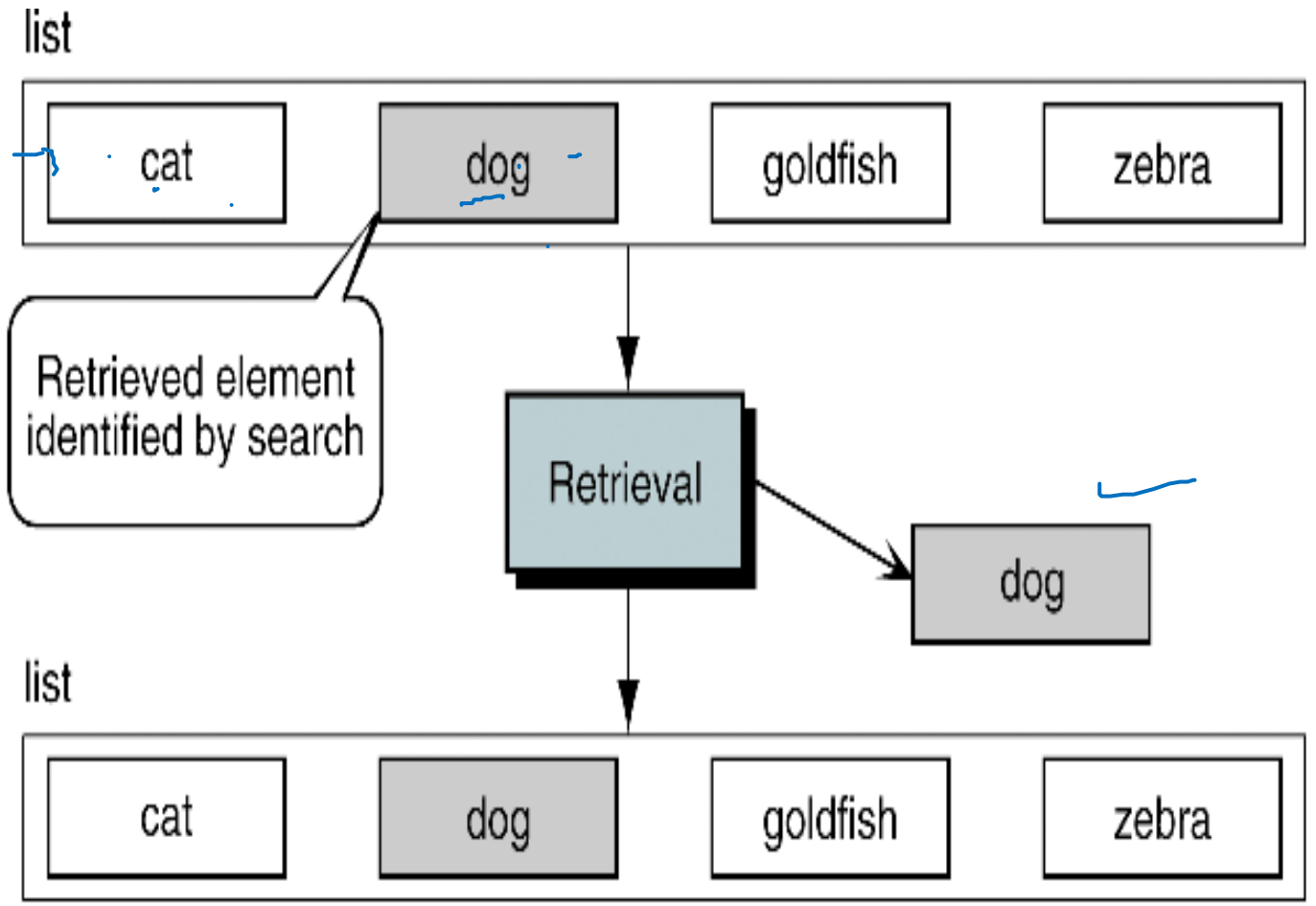


FIGURE 5-3 Retrieval

Retrieval is used to get the information related to an element without changing the structure of the list.

Traversal.

- List traversal **processes each element** in a list in sequence.

Implementation.

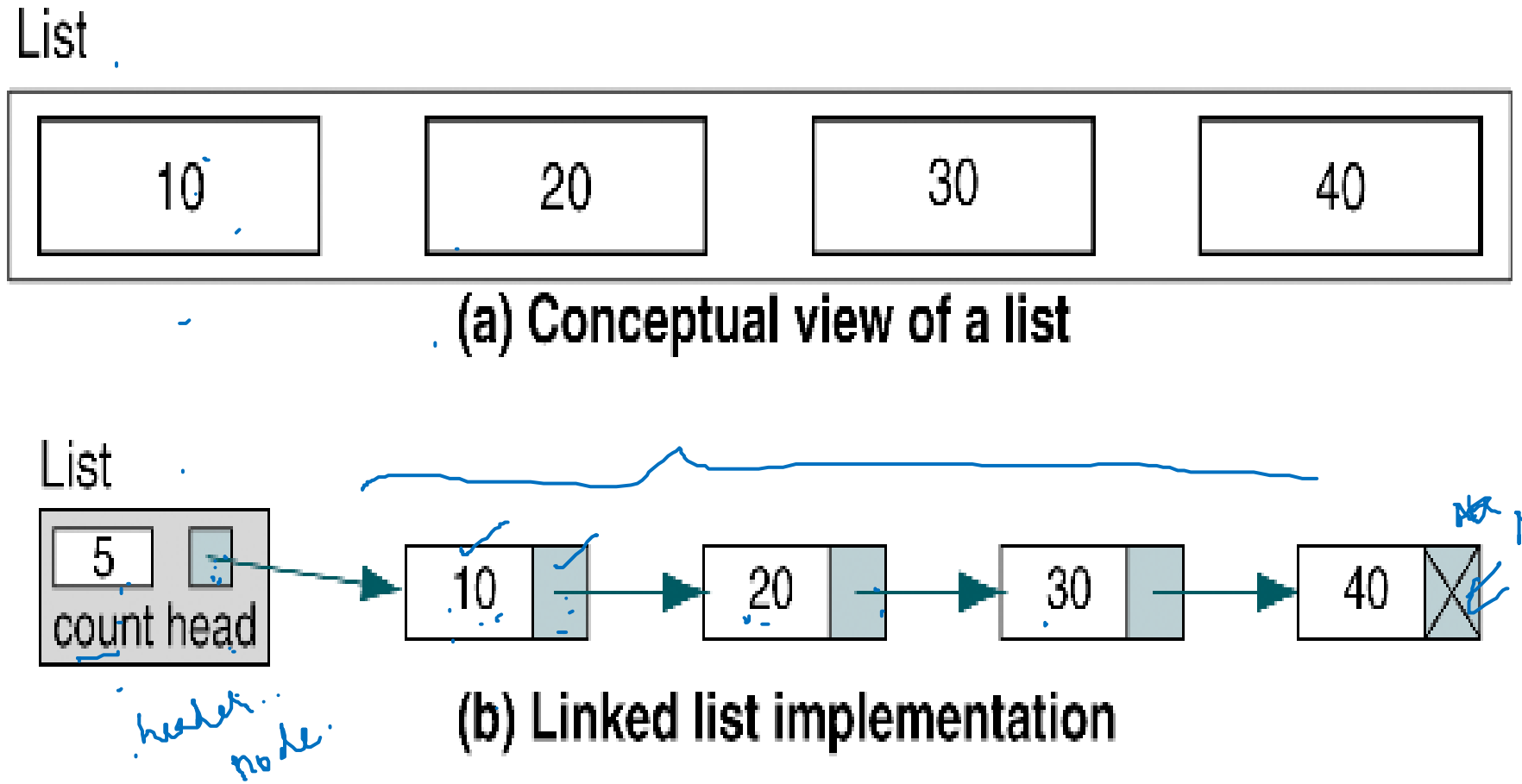
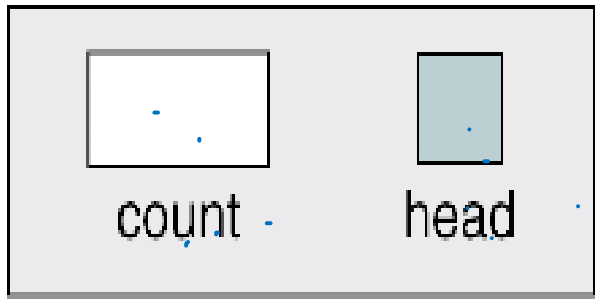
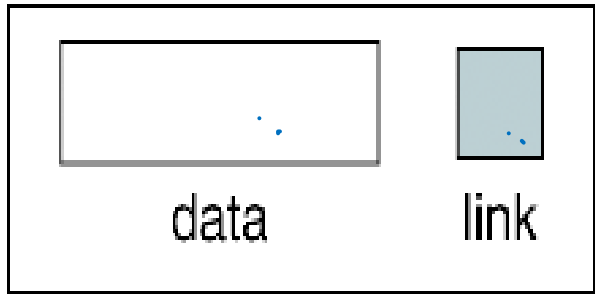


FIGURE 5-4 Linked List Implementation of a List

Data structure.



(a) Head structure



(b) Data node structure

```
list
  count
  head
end list


node
  data
  link
end node
```

FIGURE 5-5 Head Node and Data Node

Algorithms.

1. Create list
2. Insert node
3. Delete node
4. List search
5. Retrieve node
6. Empty list
7. Full list
8. List count
9. Traverse list
10. Destroy list

Create list.

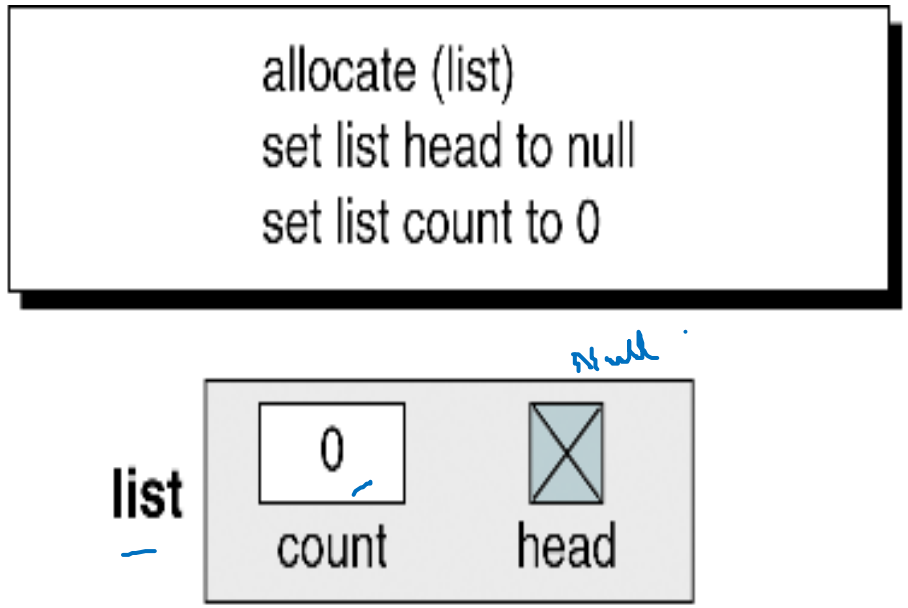


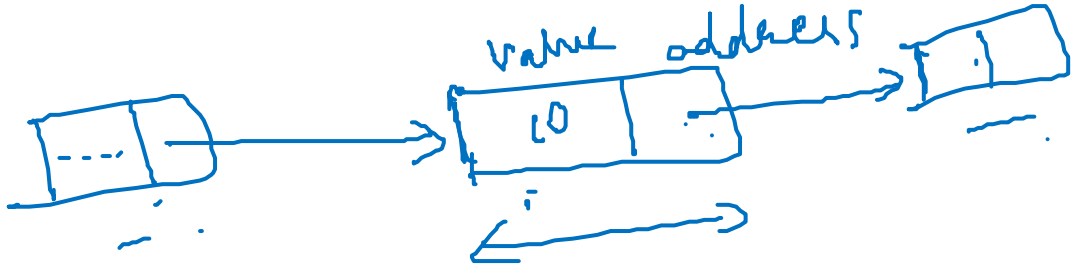
FIGURE 5-6 Create List

Create list.

ALGORITHM 5-1 Create List

```
Algorithm createList (list)
Initializes metadata for list.
  Pre    list is metadata structure passed by reference
  Post   metadata initialized
1 allocate (list)
2 set list head  to null
3 set list count to 0
end createList
```

Insert node.



- Only its **logical predecessor** is needed.
- There are three steps to the insertion:
  - **Allocate memory for the new node** and move data to the node.
  - Point the **new node to its successor**.
  - Point the **new node's predecessor to the new node**.

Insert node.

1. Insert into empty list
2. Insert at beginning
3. Insert in middle
4. Insert at end

Insert into empty list.

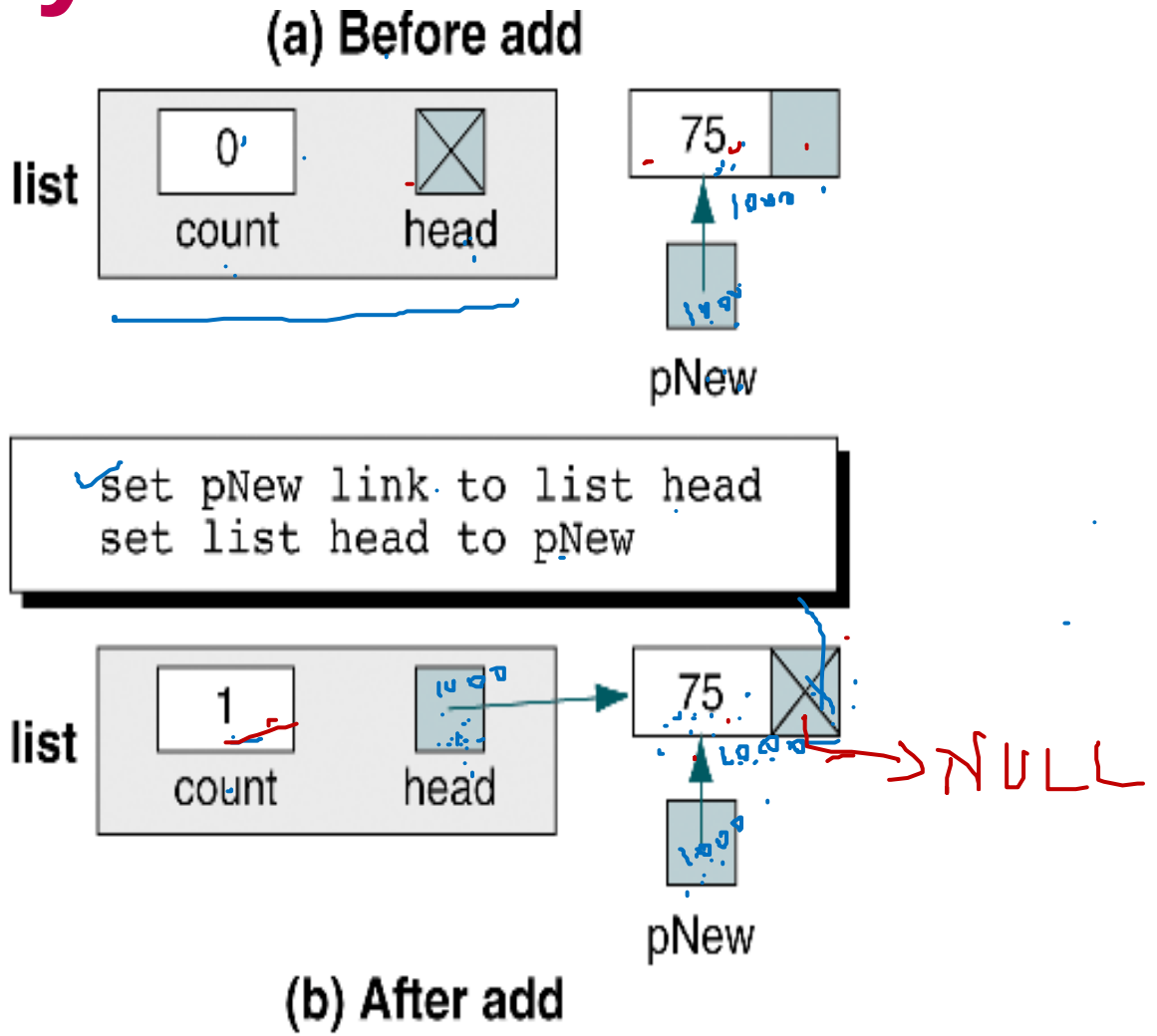


FIGURE 5-7 Add Node to Empty List

Insert at beginning.

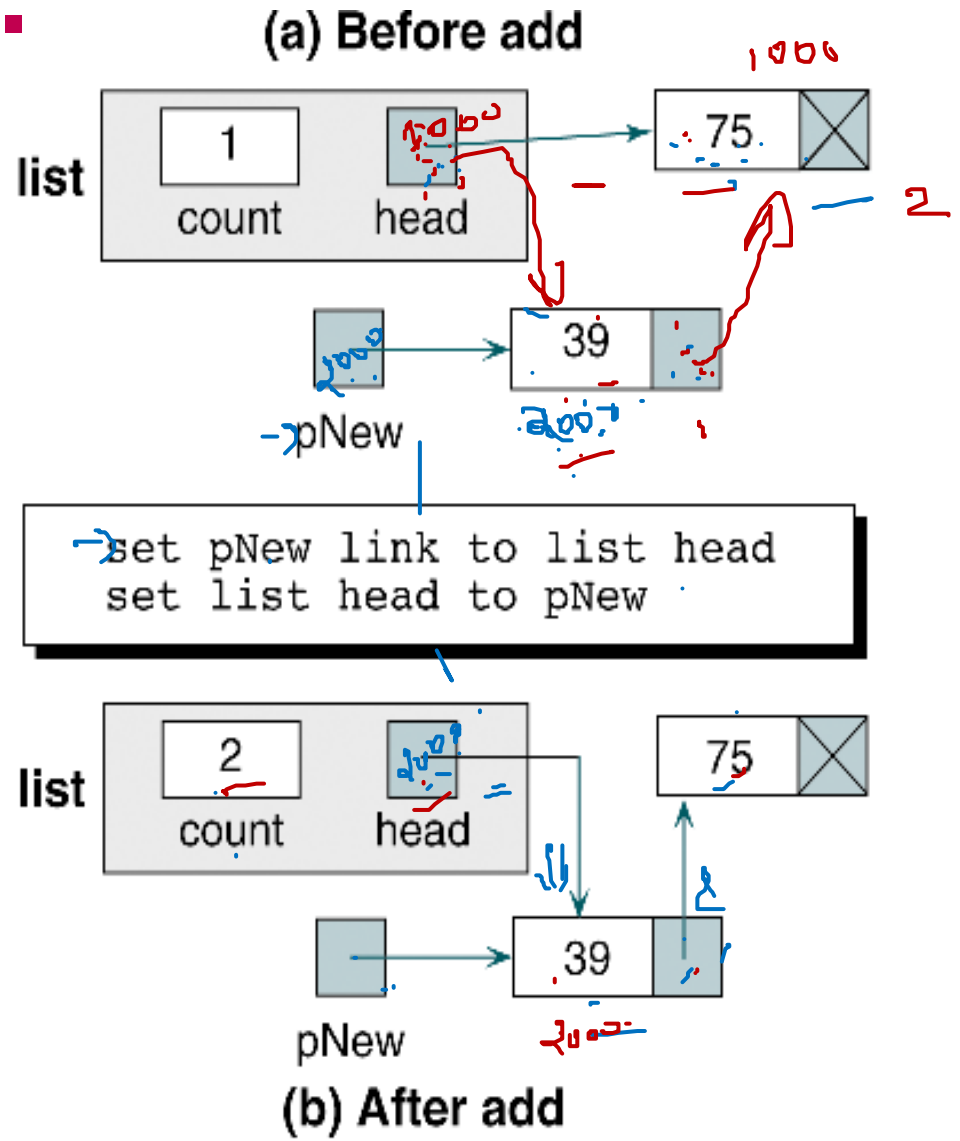


FIGURE 5-8 Add Node at Beginning

Insert in middle.

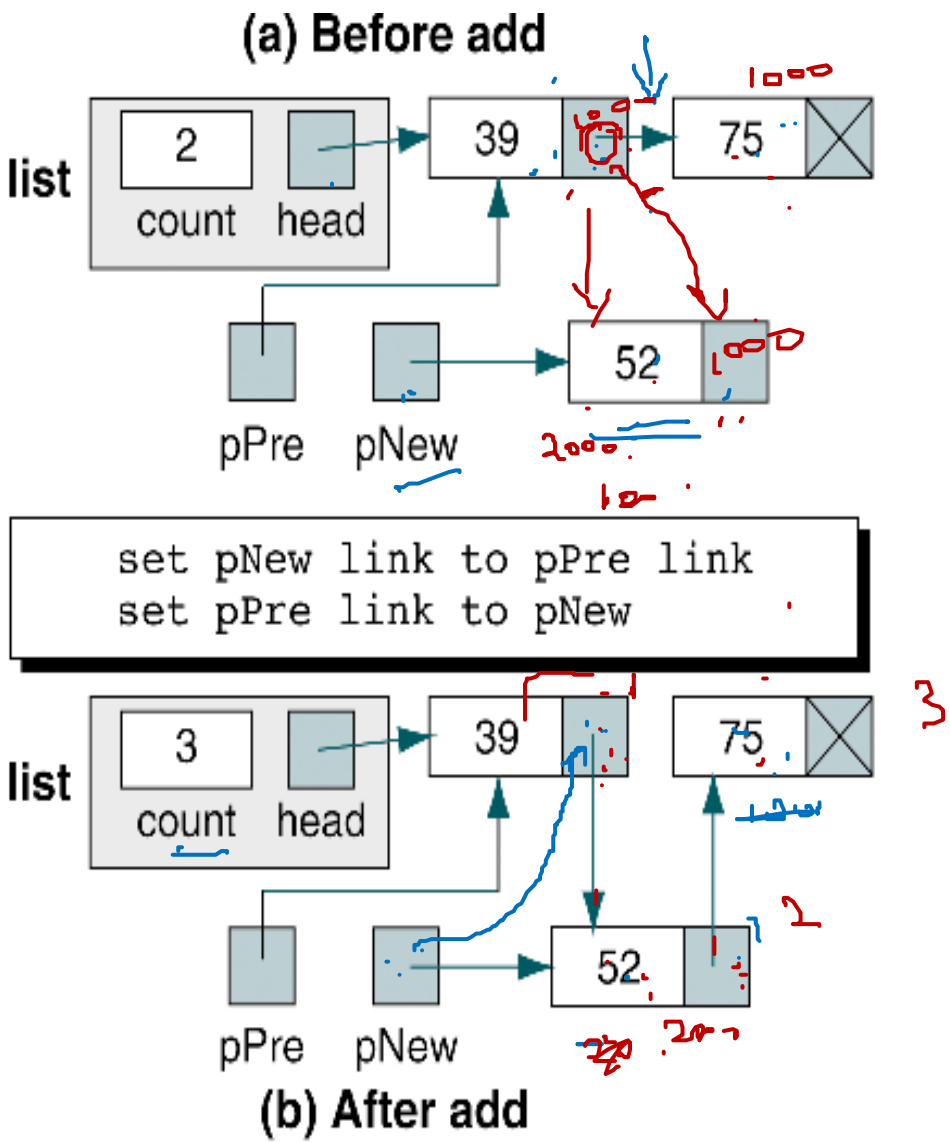


FIGURE 5-9 Add Node in Middle

Insert at end.

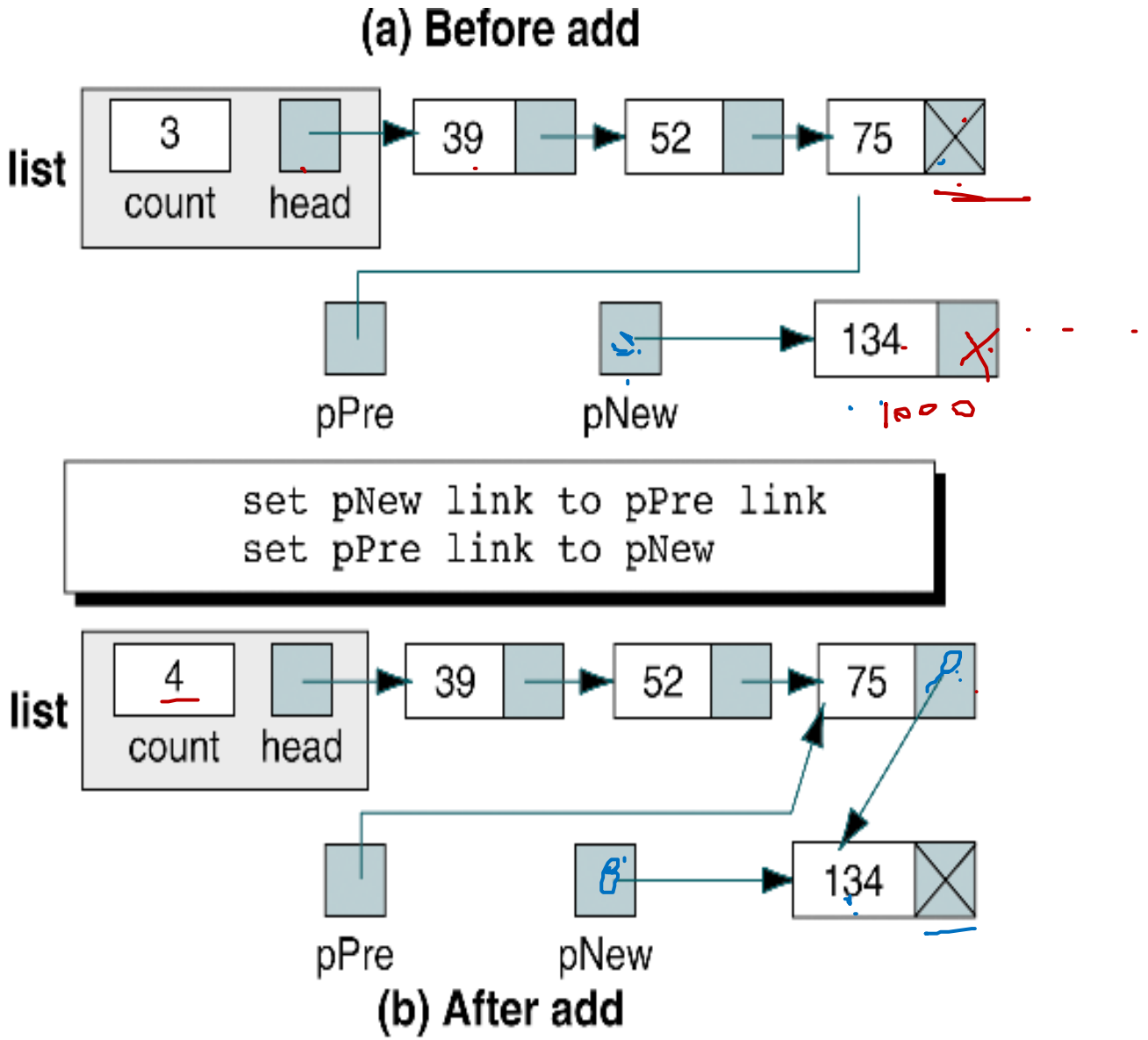


FIGURE 5-10 Add Node at End

ALGORITHM 5-2 Insert List Node

```
Algorithm insertNode (list, pPre, dataIn)
Inserts data into a new node in the list.
  Pre    list is metadata structure to a valid list
         pPre is pointer to data's logical predecessor
         dataIn contains data to be inserted
  Post   data have been inserted in sequence
  Return true if successful, false if memory overflow
1 allocate (pNew)
2 set pNew data to dataIn
3 if (pPre null)
  Adding before first node or to empty list.
  1  set pNew link to list head
  2  set list head to pNew
4 else
  Adding in middle or at end.
  1  set pNew link to pPre link
  2  set pPre link to pNew
5 end if
6 return true
end insertNode
```

Delete node.

1. Delete first node

2. General delete case

Delete first node

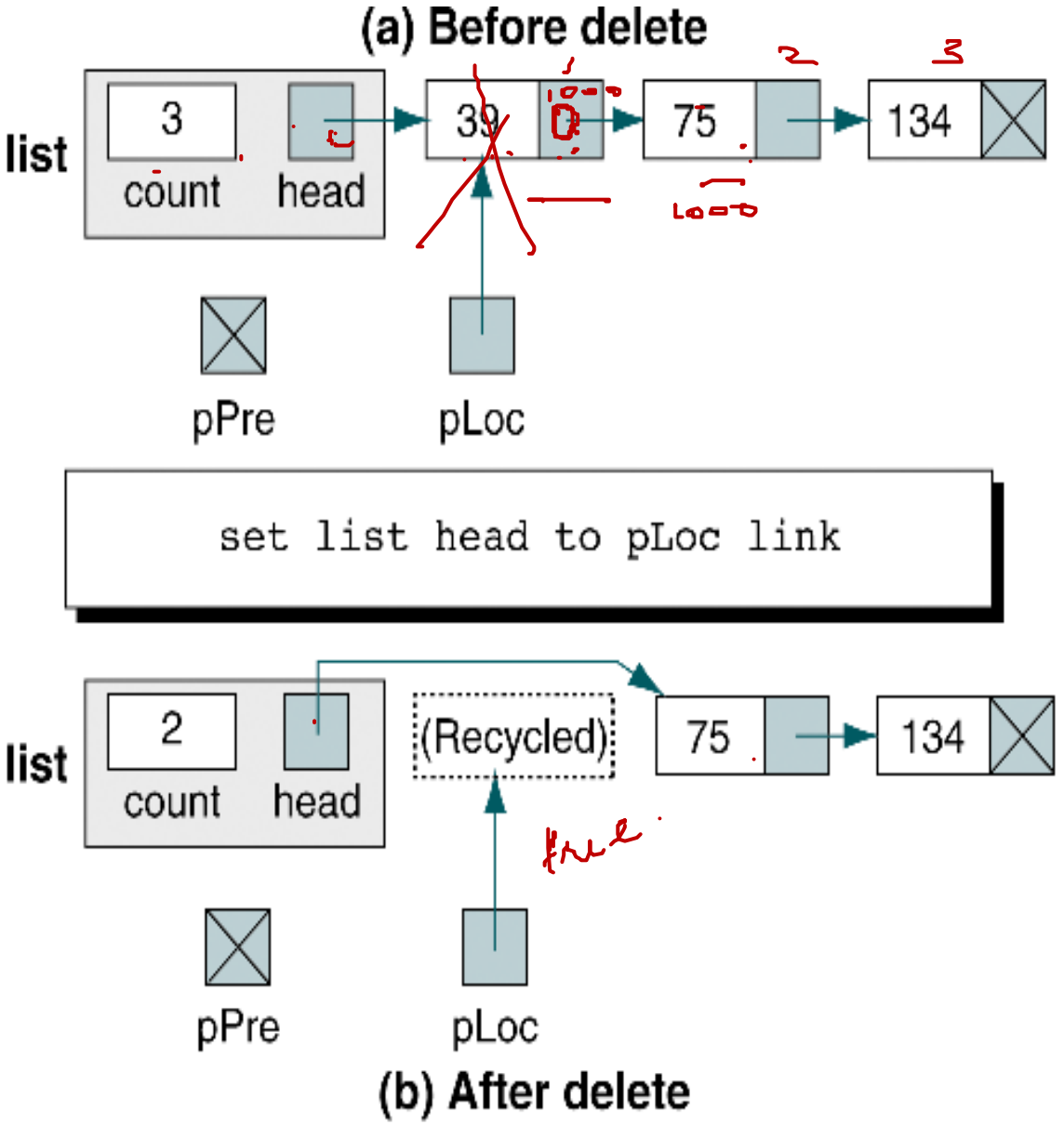


FIGURE 5-11 Delete First Node

Delete general case.

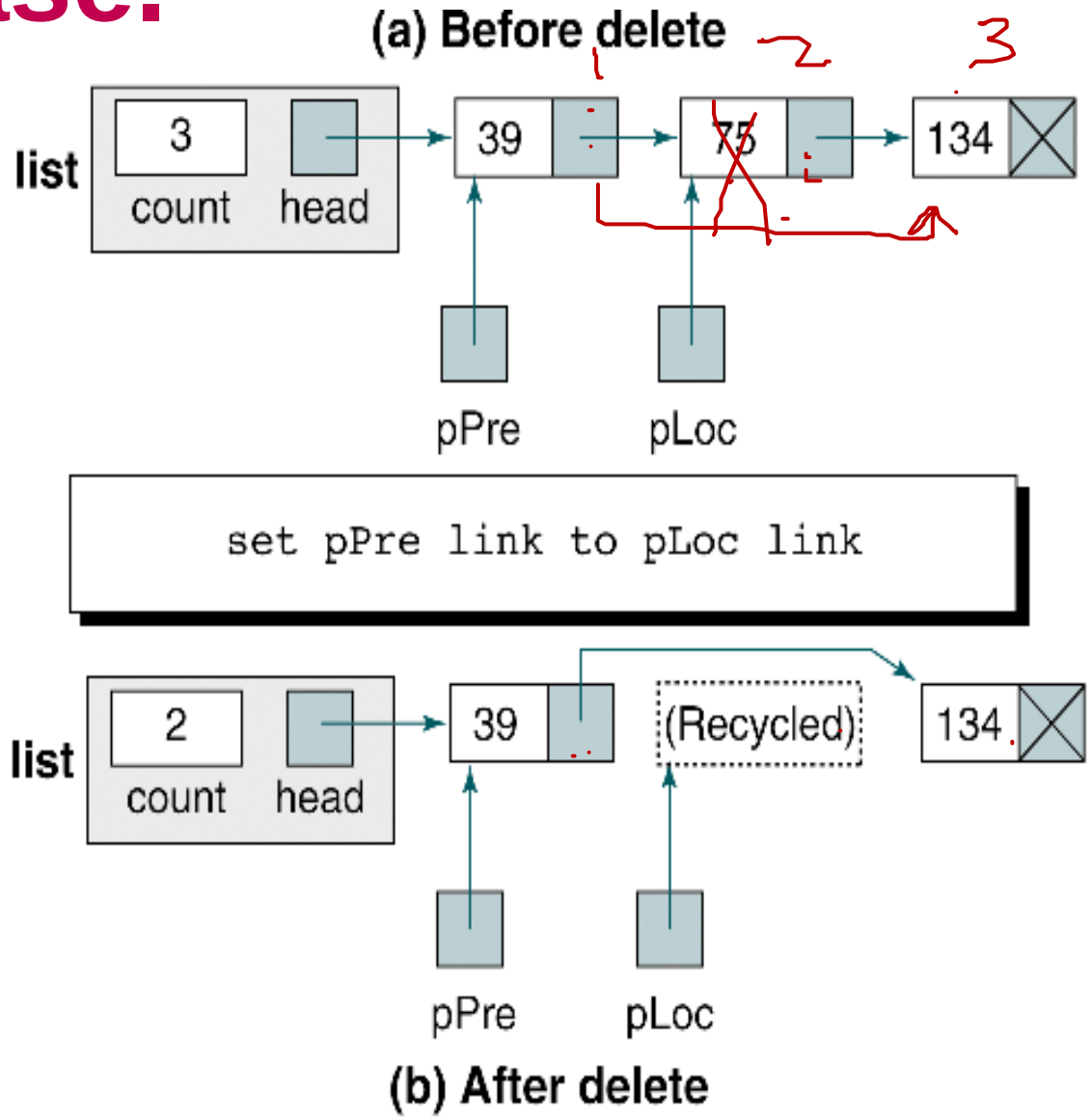


FIGURE 5-12 Delete General Case

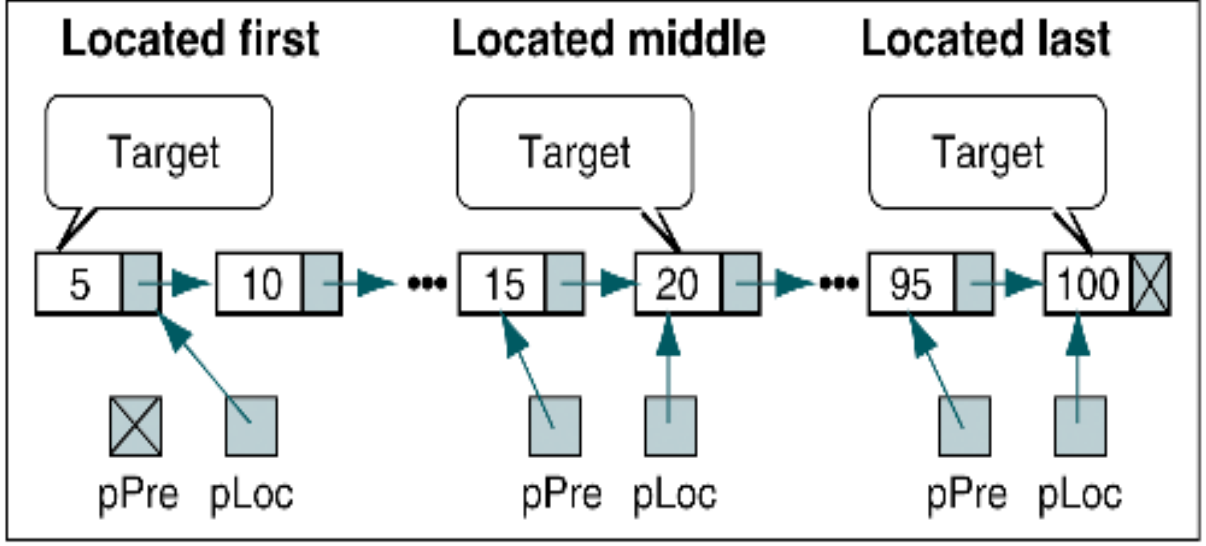
ALGORITHM 5-3 List Delete Node

```
Algorithm deleteNode (list, pPre, pLoc, dataOut)
Deletes data from list & returns it to calling module.
  Pre  list is metadata structure to a valid list
       Pre is a pointer to predecessor node
       pLoc is a pointer to node to be deleted
       dataOut is variable to receive deleted data
  Post data have been deleted and returned to caller
1 move pLoc data to dataOut
2 if (pPre null)
  Deleting first node
  1  set list head to pLoc link
3 else
  Deleting other nodes
  1  set pPre link to pLoc link
4 end if
5 recycle (pLoc)
end deleteNode
```

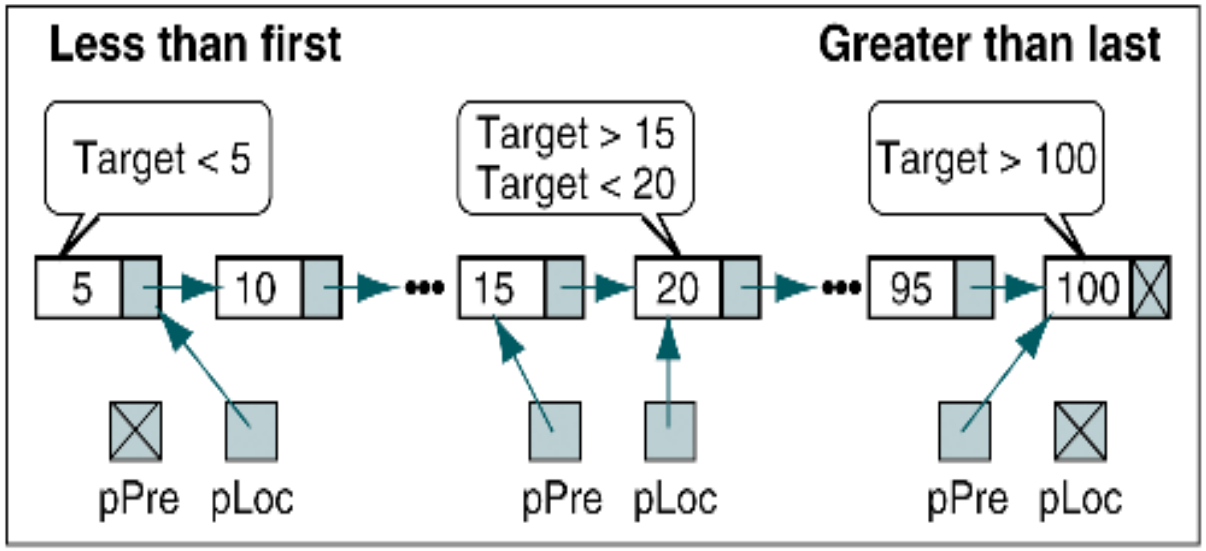
List search.

Condition	pPre	pLoc	Return
Target < first node	Null	First node	False
Target = first node	Null	First node	True
First < target < last	Largest node < target	First node > target	False
Target = middle node	Node's predecessor	Equal node	True
Target = last node	Last's predecessor	Last node	True
Target > last node	Last node	Null	False

TABLE 5-1 List Search Results



(a) Successful searches (return true)



(b) Unsuccessful searches (return false)

FIGURE 5-13 Ordered List Search

ALGORITHM 5-4 Search List

```
Algorithm searchList (list, pPre, pLoc, target)
Searches list and passes back address of node containing
target and its logical predecessor.
  Pre   list is metadata structure to a valid list
        pPre is pointer variable for predecessor
        pLoc is pointer variable for current node
        target is the key being sought
  Post  pLoc points to first node with equal/greater key
        -or- null if target > key of last node
        pPre points to largest node smaller than key
        -or- null if target < key of first node
  Return true if found, false if not found
1 set pPre to null
2 set pLoc to list head
3 loop (pLoc not null AND target > pLoc key)
  1  set pPre to pLoc
  2  set pLoc to pLoc link
4 end loop
5 if (pLoc null)
  Set return value
  1  set found to false
```

continued

ALGORITHM 5-4 Search List *(continued)*

```
 6 else
   1  if (target equal pLoc key)
      1  set found to true
   2  else
      1  set found to false
   3  end if
 7 end if
 8 return found
end searchList
```

Retrieve node.

ALGORITHM 5-5 Retrieve List Node

```
Algorithm retrieveNode (list, key, dataOut)
Retrieves data from a list.
  Pre    list is metadata structure to a valid list
         key is target of data to be retrieved
         dataOut is variable to receive retrieved data
  Post   data placed in dataOut
          -or- error returned if not found
  Return true if successful, false if data not found
1 set found to searchList (list, pPre, pLoc, key)
2 if (found)
  1  move pLoc data to dataOut
3 end if
4 return found
end retrieveNode
```

Empty list.

ALGORITHM 5-6 Empty List

```
Algorithm emtpyList (list)
Returns Boolean indicating whether the list is empty.
  Pre    list is metadata structure to a valid list
  Return true if list empty, false if list contains data
1 if (list count equal 0)
  1  return true
2 else
  1  return false
end emptyList
```

Full list.

ALGORITHM 5-7 Full List

```
Algorithm fullList (list)
Returns Boolean indicating whether or not the list is full.
  Pre    list is metadata structure to a valid list
  Return false if room for new node; true if memory full
1 if (memory full)
  1  return true
2 else
  2  return false
3 end if
4 return true
end fullList
```

List count.

ALGORITHM 5-8 List Count

```
Algorithm listCount (list)
Returns integer representing number of nodes in list.
  Pre    list is metadata structure to a valid list
  Return count for number of nodes in list
 1 return (list count)
end listCount
```

Traversal list.

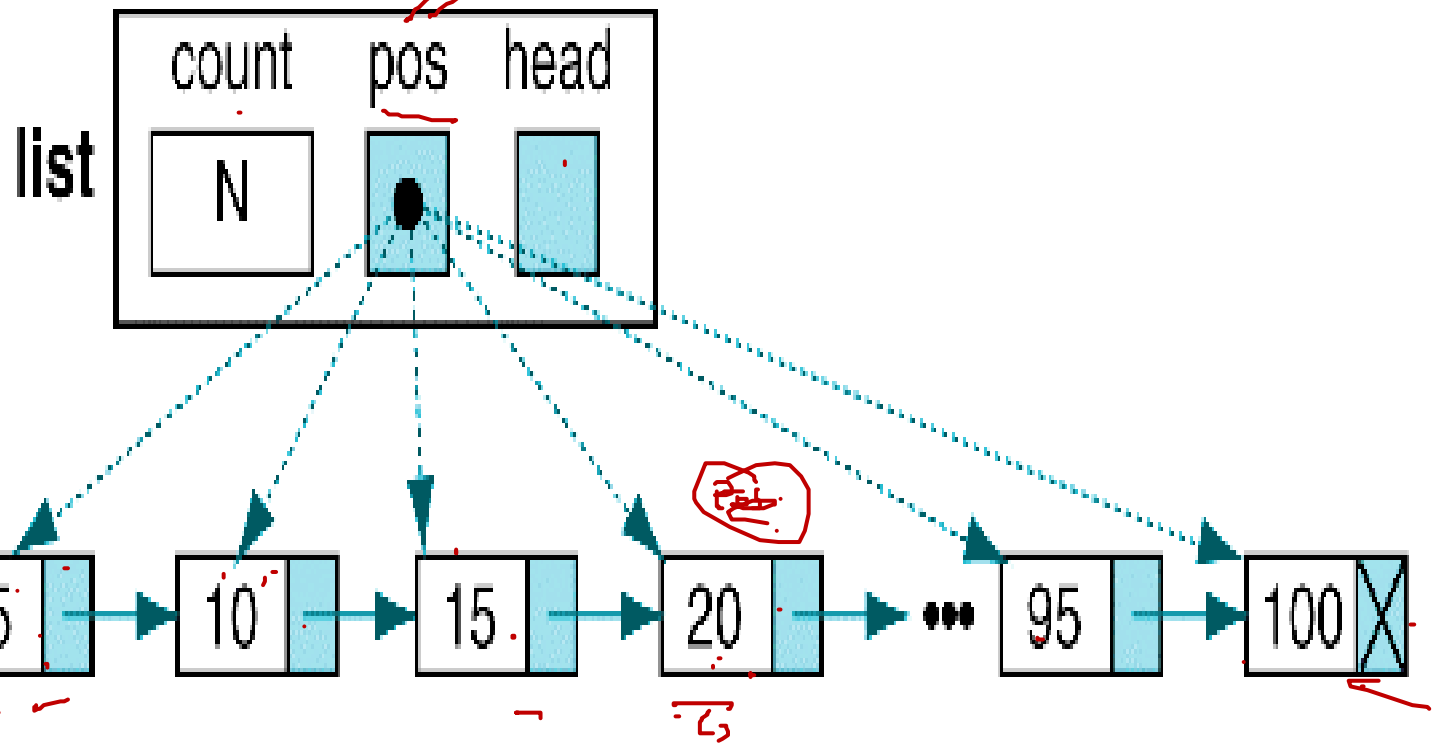


FIGURE 5-14 List Traversal

ALGORITHM 5-9 Traverse List

```
Algorithm getNext (list, fromWhere, dataOut)
Traverses a list. Each call returns the location of an
element in the list.
  Pre    list is metadata structure to a valid list
         fromWhere is 0 to start at the first element
         dataOut is reference to data variable
  Post   dataOut contains data and true returned
         -or- if end of list, returns false
```

continued

ALGORITHM 5-9 Traverse List *(continued)*

```
   Return true if next element located
          false if end of list
 1 if (empty list)
   1  return false
 2 if (fromWhere is beginning)
   Start from first
   1  set list pos to list head
   2  move current list data to dataOut
   3  return true
 3 else
   Continue from pos
   1  if (end of list)
      End of List
      1  return false
    2 else
      1  set list pos to next node
      2  move current list data to dataOut
      3  return true
    3 end if
 4 end if
end getNext
```

Destroy list.

ALGORITHM 5-10 Destroy List

```
Algorithm destroyList (pList)
Deletes all data in list.
  Pre    list is metadata structure to a valid list
```

continued

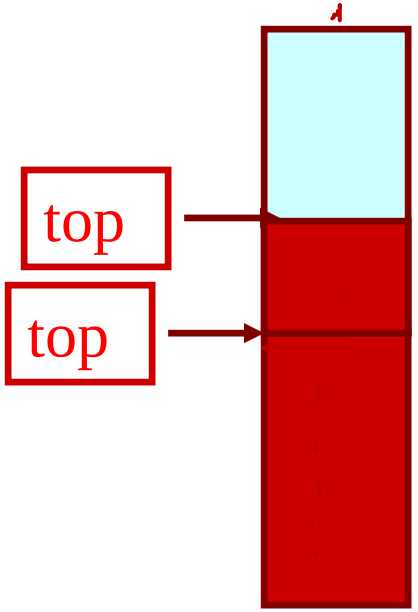
ALGORITHM 5-10 Destroy List *(continued)*

```
  Post   All data deleted
1 loop (not at end of list)
  1  set list head to successor node
  2  release memory to heap
2 end loop
  No data left in list. Reset metadata.
3 set list pos to null
4 set list count to 0
end destroyList
```

STACK IMPLEMENTATIONS: USING ARRAY AND LINKED LIST

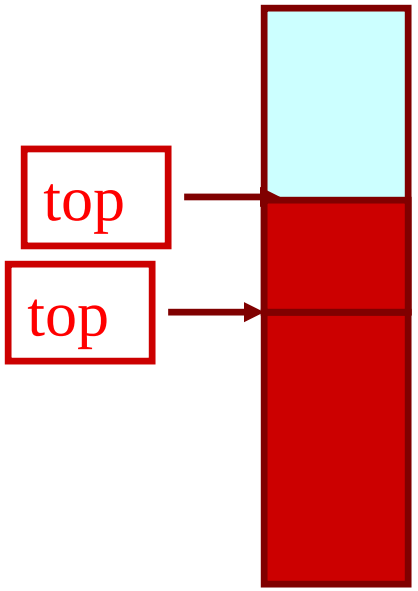
STACK USING ARRAY

PUSH



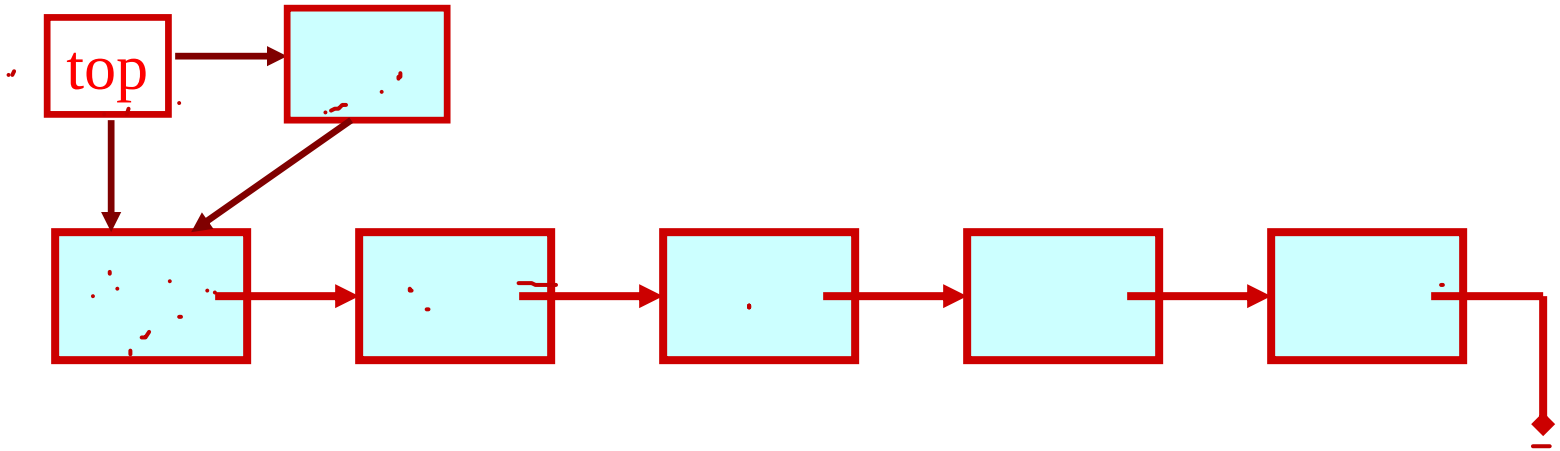
STACK USING ARRAY

POP



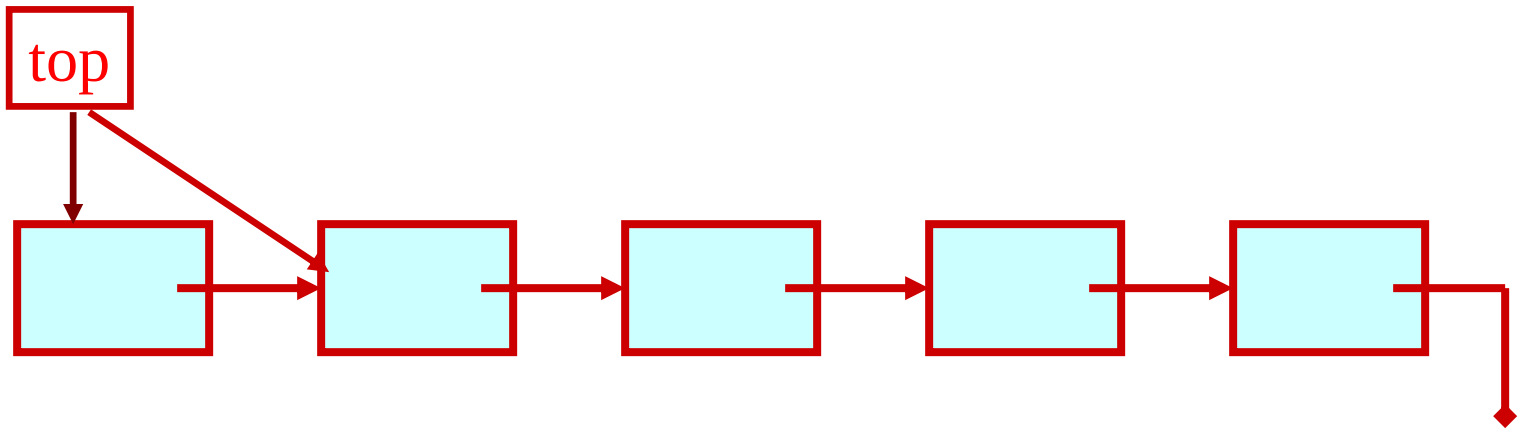
Stack: Linked List Structure

PUSH OPERATION

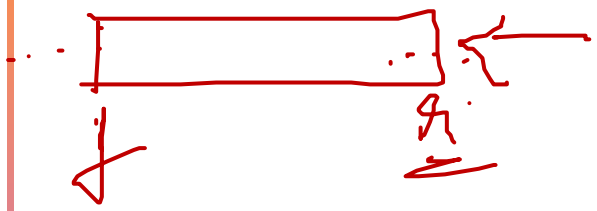


Stack: Linked List Structure

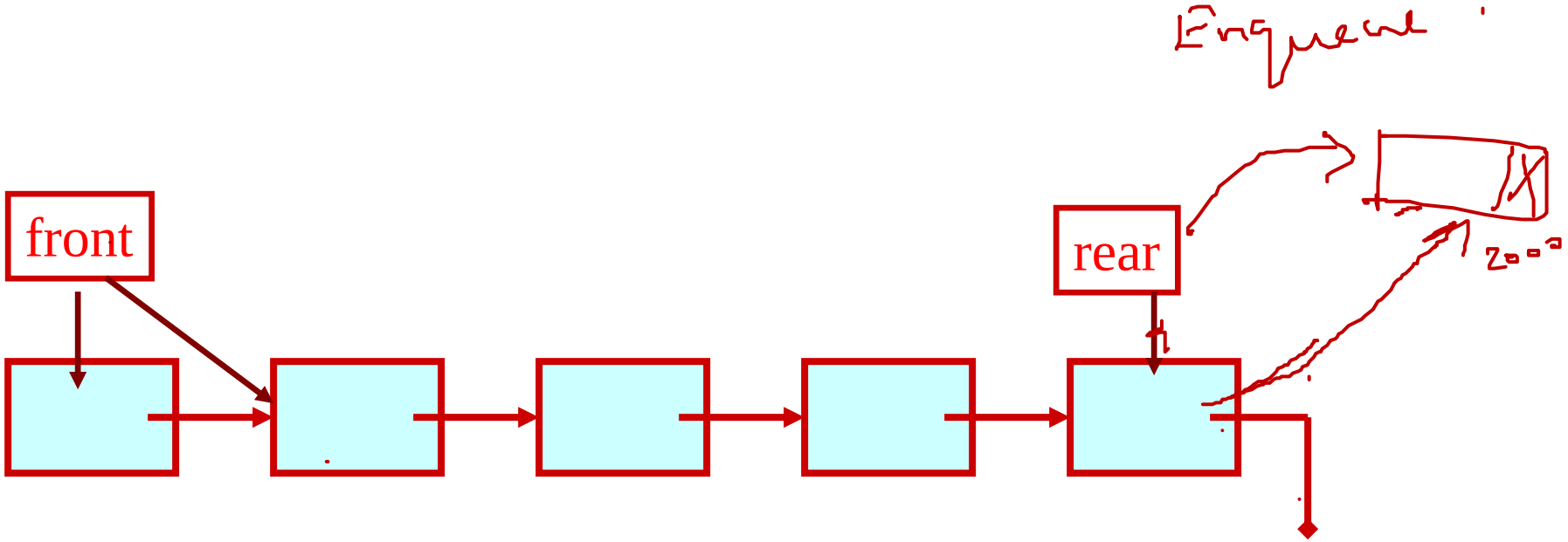
POP OPERATION



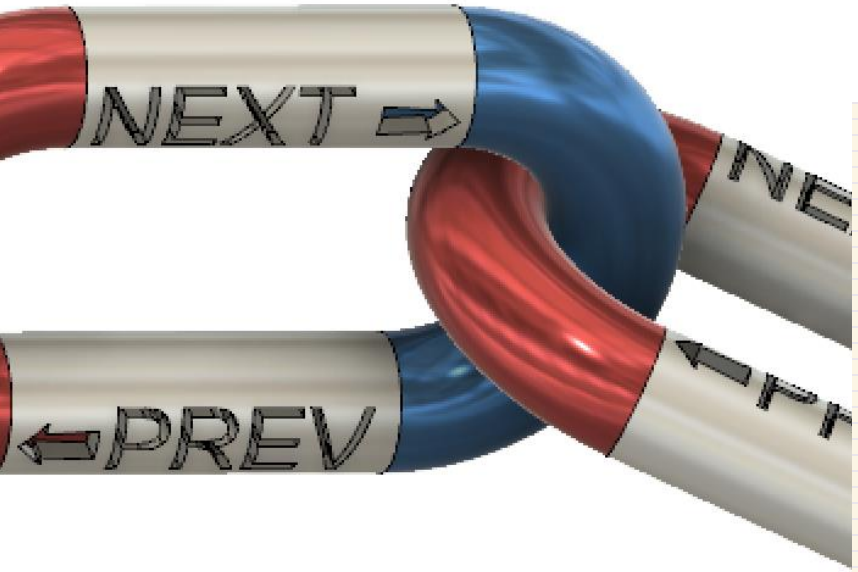
QUEUE: LINKED LIST STRUCTURE



DEQUEUE

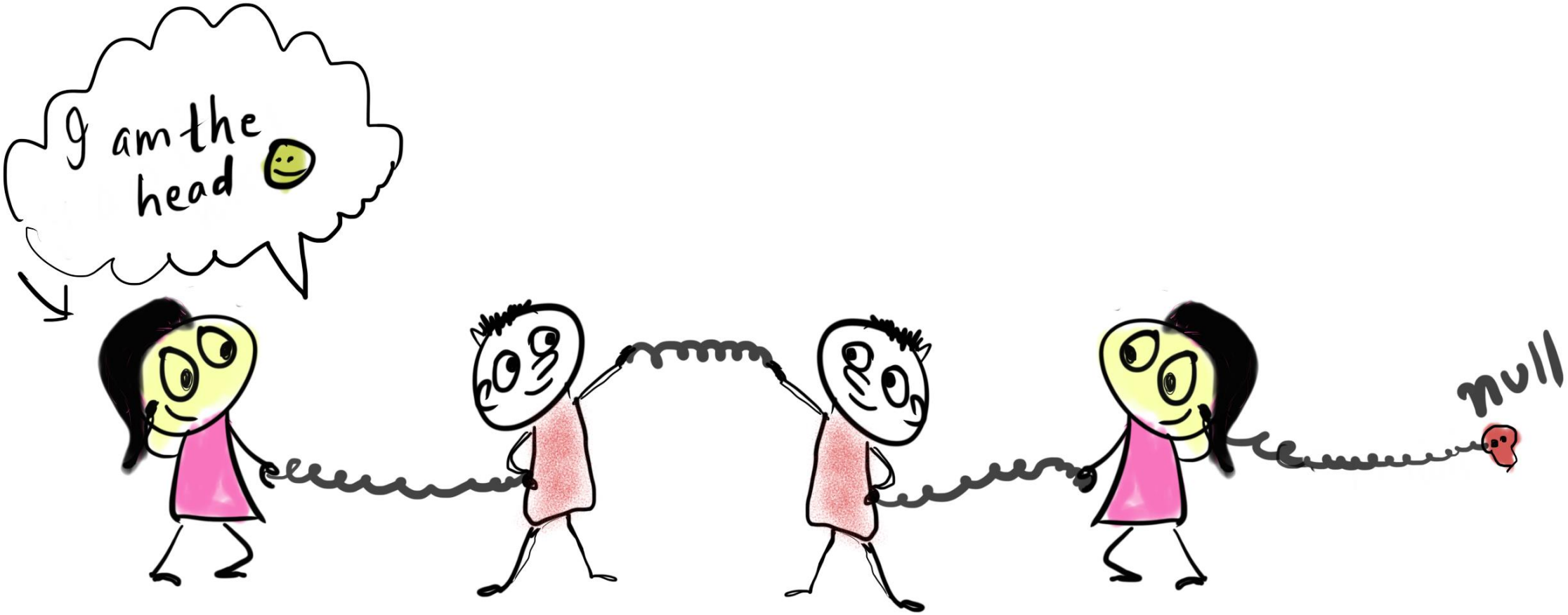


THE END.



Doubly and Circular Linked Lists

SINGLY LINKED LIST.



RECAP TO GO FORWARD.

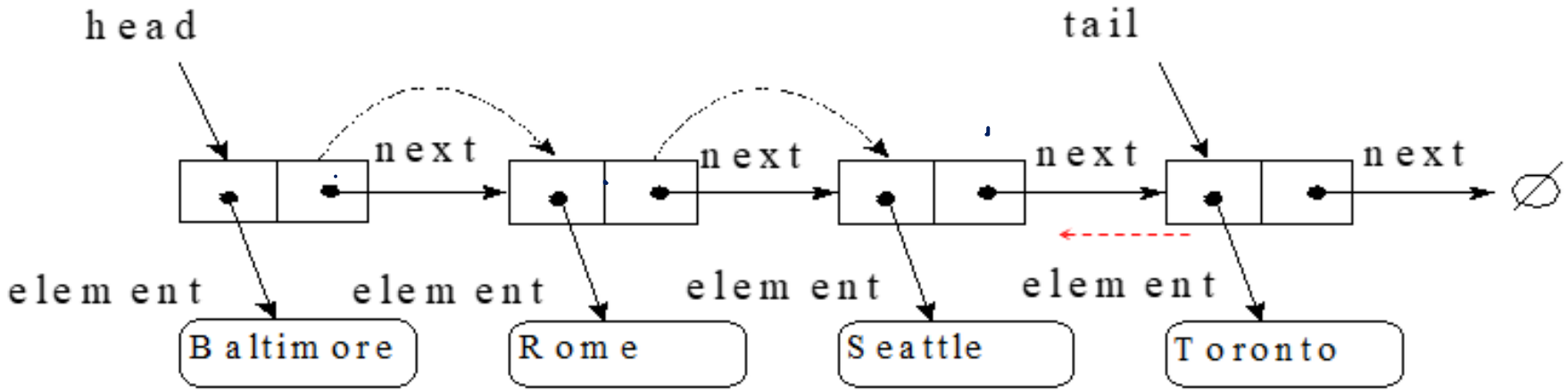
- The node at the beginning is called the **head** of the list
- The entire list is identified by the pointer to the head node, this pointer is called the **list head**.
- Nodes can be **added or removed** from the linked list during execution
- Addition or removal of nodes can take place **at beginning**, **end**, or **middle** of the list
- Linked lists can **grow and shrink** as needed, unlike arrays, which have a fixed size.

TYPES OF LISTS.

- Depending on the way in which the links are used to maintain adjacency, several different types of linked lists are possible.
    - Singly Linked List
    - Doubly Linked List
    - Circular Linked List

DOUBLY LINKED LIST.

- Recall, deletion of element in singly linked list required link hopping to find last node(tail node).
- Made easier with doubly linked list.

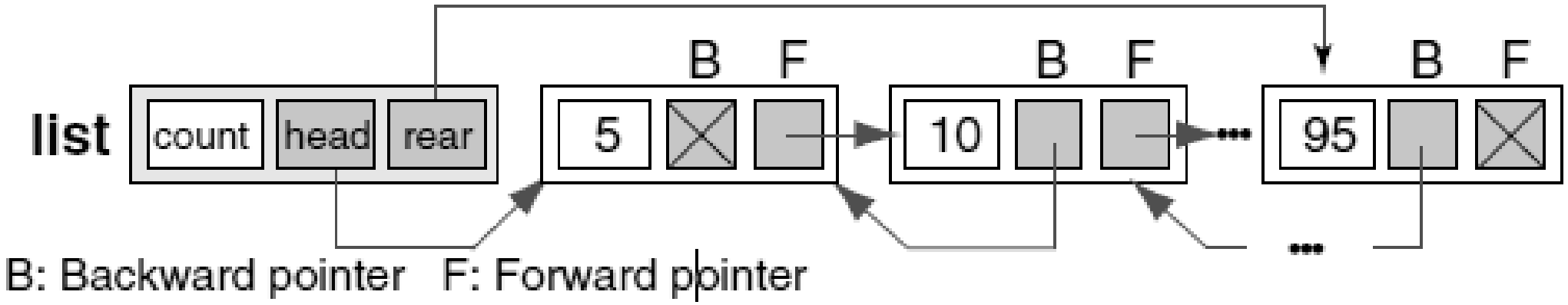


DOUBLY LINKED LIST.

- Is a linked list structure in which each node has a pointer to both its ***successor*** and its ***predecessor***.
- Pointers exist between adjacent nodes in both directions.
- The list can be traversed either forward or backward.
- There are three pieces of metadata in the head structure: **a count, a position pointer for traversals, and a rear pointer.**

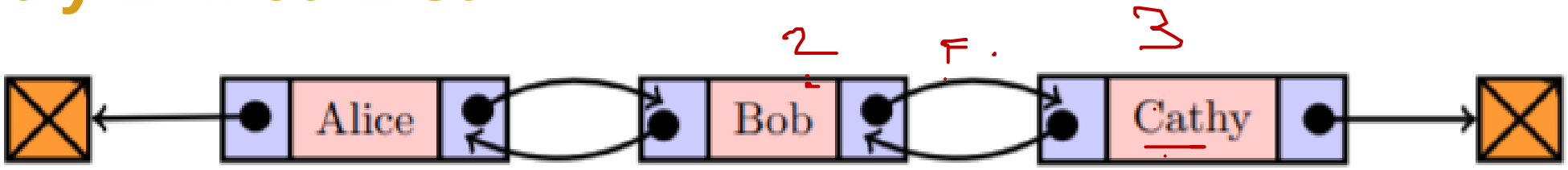
DOUBLY LINKED LIST.

- Although a rear pointer is not required in all doubly linked lists, it makes some of the list algorithms, such as insert and search, more efficient.
- Each node contains two pointers: ***a backward pointer*** to its **predecessor** and a ***forward pointer*** to its **successor.**

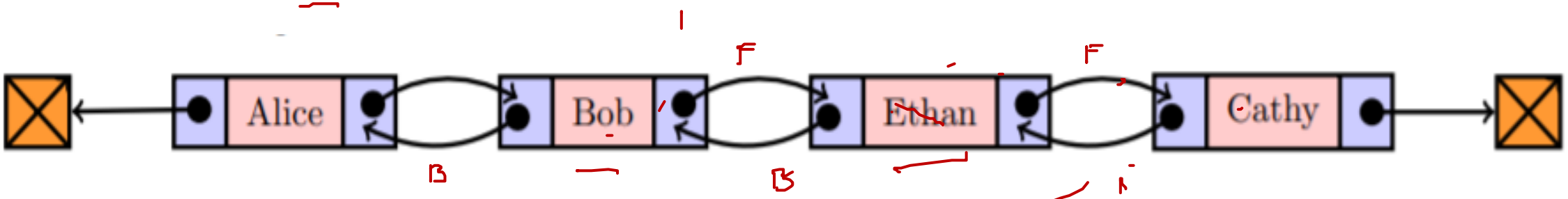


DOUBLY LINKED LIST.

Doubly Linked List



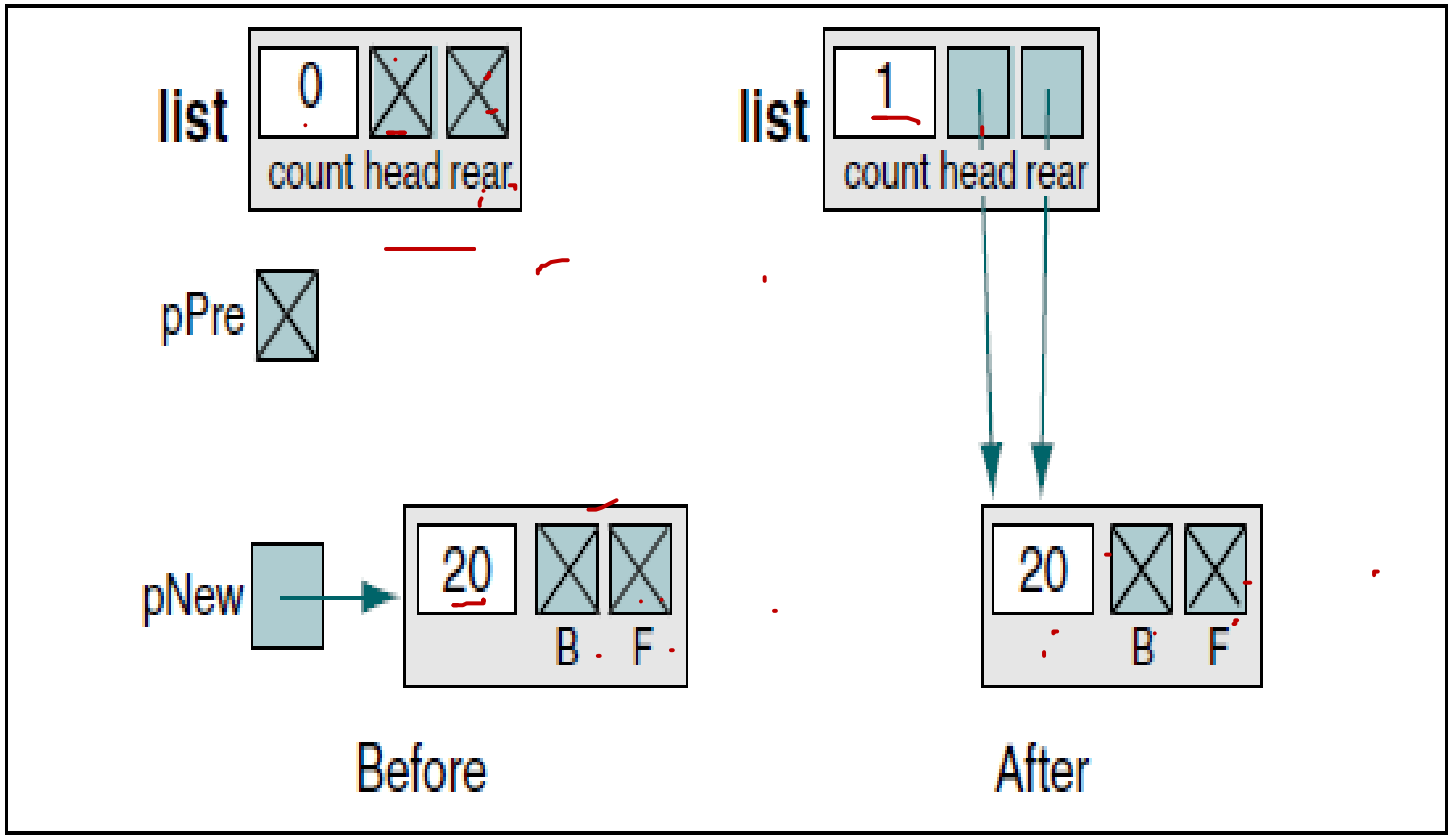
Insert a node



Delete a node

DOUBLY LINKED LIST - INSERTION.

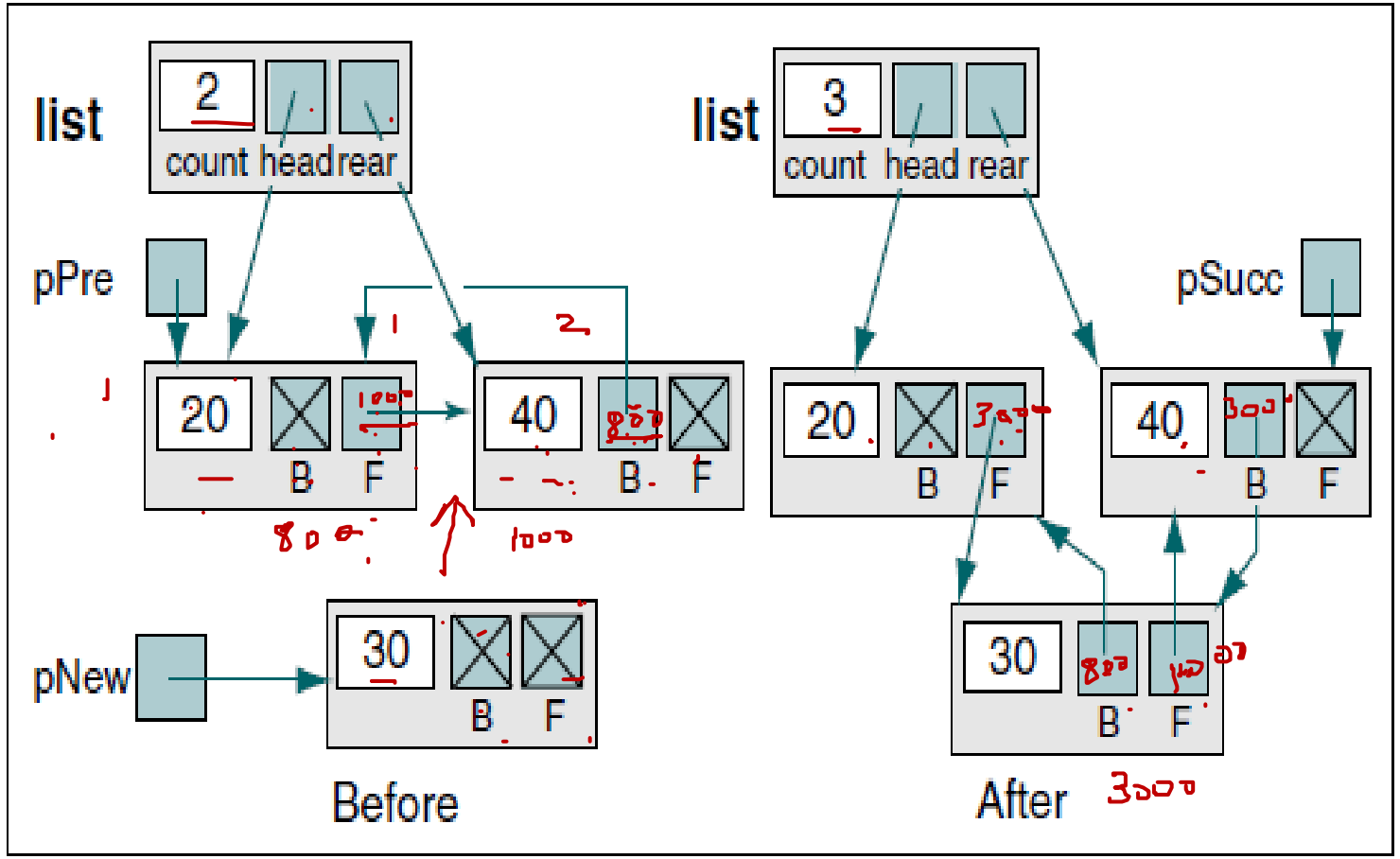
- Inserting follows the basic pattern of inserting a node into a singly linked list, but we also need to **connect both the forward and the backward** pointers.
- A null doubly linked list's **head and rear pointers are null.**
- To insert a node into a null list, we simply **set the head and rear pointers to point to the new node** and set the **forward and backward pointers of the new node to null.**



(a) Insert into null list or before first node

DOUBLY LINKED LIST - INSERTION.

- For the insertion between two nodes:
- The new node needs to be set to point to both its predecessor and its successor, and they need to be set to point to the new node.
- Because the insert is in the middle of the list, the head structure is unchanged.



(b) Insert between two nodes

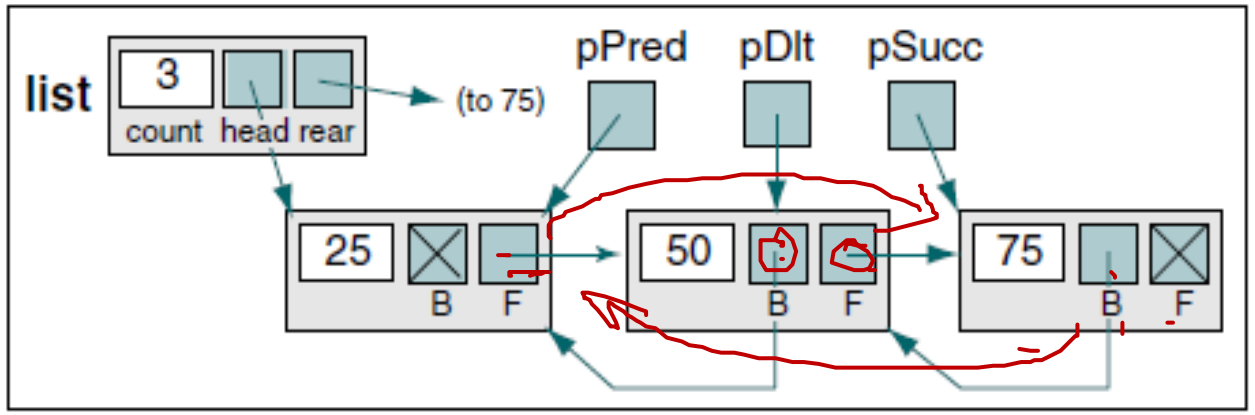
ALGORITHM 5-11 Doubly Linked List Insert

```
Algorithm insertDbl (list, dataIn)
This algorithm inserts data into a doubly linked list.
  Pre    list is metadata structure to a valid list
         dataIn contains the data to be inserted
  Post   The data have been inserted in sequence
  Return 0: failed--dynamic memory overflow
         1: successful
         2: failed--duplicate key presented
1 if (full list)
  1  return 0
2 end if
  Locate insertion point in list.
3 set found to searchList
       (list, predecessor, successor, dataIn key)
4 if (not found)
  1  allocate new node
  2  move dataIn to new node
  3  if (predecessor is null)
     Inserting before first node or into empty list
     1  set new node back pointer to null
     2  set new node fore pointer to list head
     3  set list head to new node
```

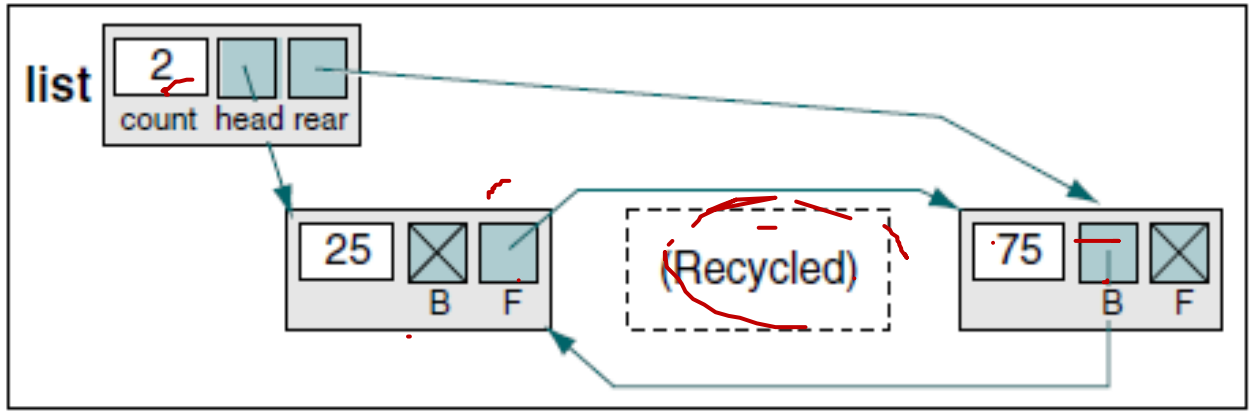
```
  4  else
       Inserting into middle or end of list
     1  set new node fore pointer to predecessor fore pointer
     2  set new node back pointer to predecessor
  5  end if
     Test for insert into null list or at end of list
  6  if (predecessor fore null)
       Inserting at end of list--set rear pointer
     1  set list rear to new node
  7  else
       Inserting in middle of list--point successor to new
     1  set successor back to new node
  8  end if
  9  set predecessor fore to new node
  10 return 1
 5 end if
   Duplicate data. Key already exists.
 6 return 2
end insertDbl
```

DOUBLY LINKED LIST - DELETION.

- Deletion of nodes:
- Deleting requires that the deleted node's predecessor, if present, be pointed to the deleted node's successor and that the successor, if present, be set to point to the predecessor.



(a) Before delete



(b) After deleting 50

- Once we locate the node to be deleted, we simply change its predecessor's and successor's pointers and recycle the node

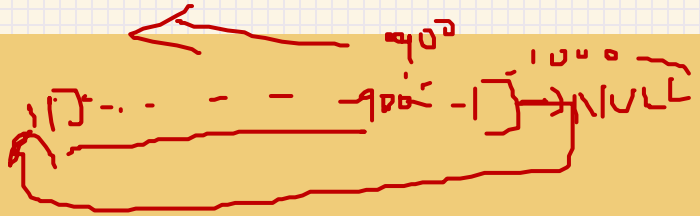
Doubly Linked List Delete

ALGORITHM 5-12 Doubly Linked List Delete

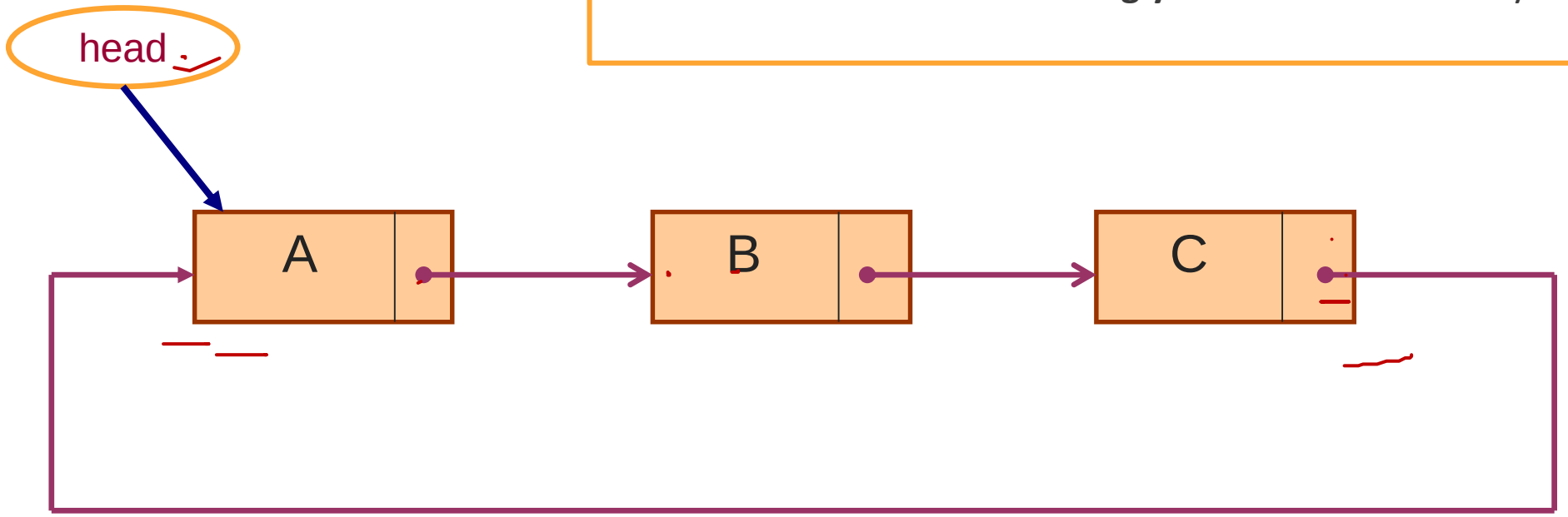
```
Algorithm deleteDbl (list, deleteNode)
This algorithm deletes a node from a doubly linked list.
   Pre    list is metadata structure to a valid list
          deleteNode is a pointer to the node to be deleted
   Post   node deleted
 1 if (deleteNode null)
   1  abort ("Impossible condition in delete double")
 2 end if
 3 if (deleteNode back not null)
   Point predecessor to successor
   1  set predecessor      to deleteNode back
   2  set predecessor fore to deleteNode fore
 4 else
   Update head pointer
   1  set list head to deleteNode fore
 5 end if
```

```
6 if (deleteNode fore not null)
  Point successor to predecessor
  1  set successor       to deleteNode fore
  2  set successor back to deleteNode back
7 else
  Point rear to predecessor
  1  set list rear to deleteNode back
8 end if
9 recycle (deleteNode)
end deleteDbl
```

CIRCULAR LINKED LIST.



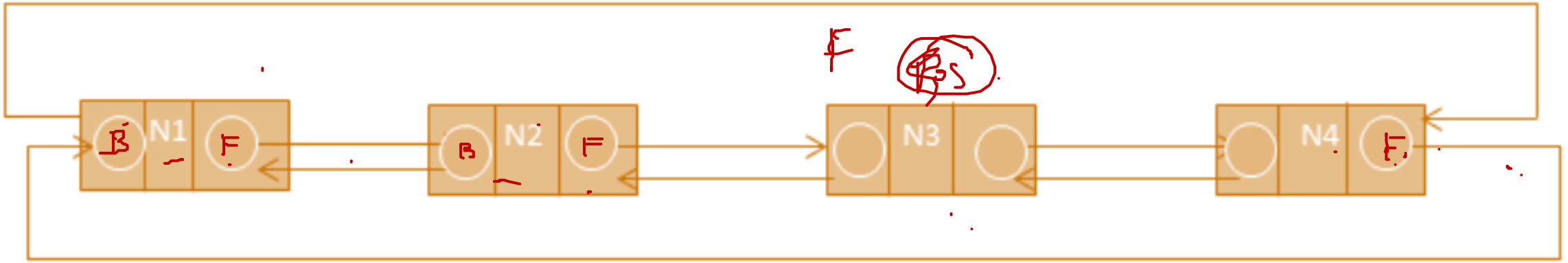
A circular linked list can be a **singly linked list** or a doubly linked list.



In ***singly linked list***, pointer from the last element in the list points back to the first element

CIRCULAR LINKED LIST.

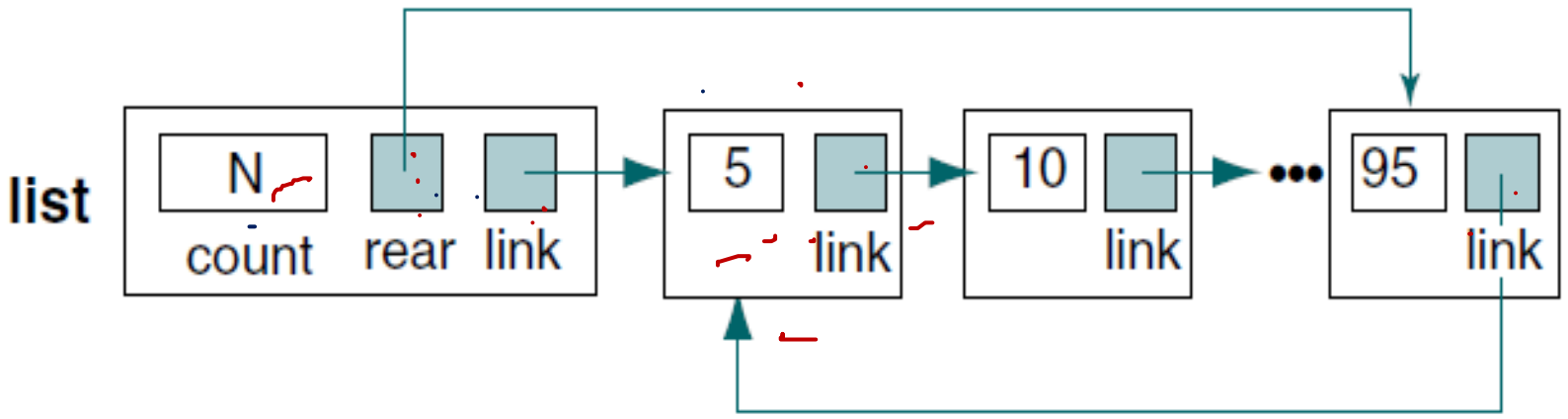
A circular linked list can be a singly linked list or a **doubly linked list.**



In a ***doubly circular linked list***, the previous pointer of the first node is connected to the last node while the next pointer of the last node is connected to the first node.

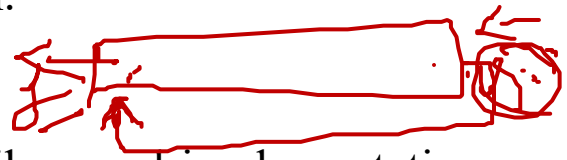
CIRCULAR LINKED LIST.

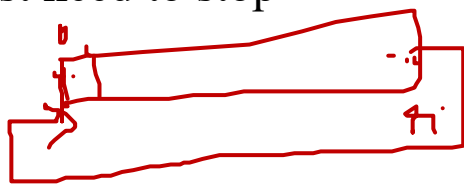
- **Circularly linked lists are primarily used in lists that allow access to nodes in the middle of the list without starting at the beginning.**
- Insertion into and deletion from a circularly linked list follow the same logic patterns used in a singly linked list except that the last node points to the first node. So, when inserting or deleting the last node, in addition to updating the rear pointer in the header, the link field must point to the first node.



Advantages of Circular Linked Lists:

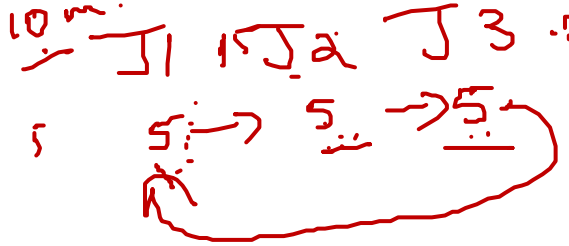
1. **Any node can be a starting point**. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.





2. Useful for implementation of queue. Unlike usual implementation, **we don't need to maintain two pointers for front and rear if we use circular linked list**. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.

3. **Circular lists are useful in applications to repeatedly go around the list**. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the **operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.**



The end.