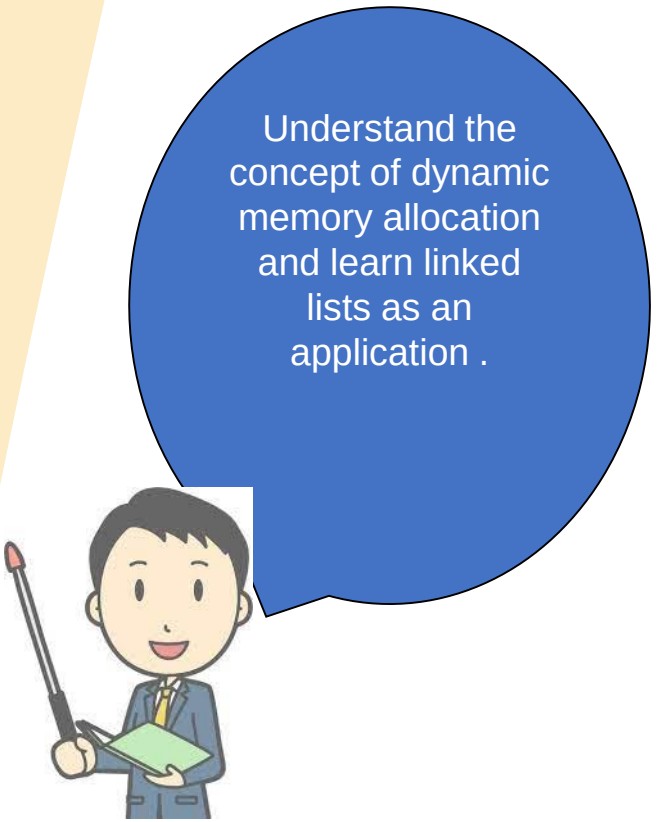
Ch.17 Dynamic Memory and Linked Lists

What you will learn in this chapter

- Understanding dynamic memory allocation
- Functions related to dynamic memory allocation
- linked list



Concept of dynamic allocation memory

- How a program allocates memory
  - Static allocation
  - dynamic allocation

Static memory allocation

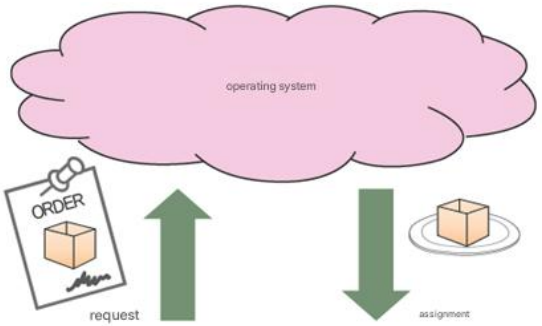
- Static memory allocation
  - Allocating a predetermined amount of memory before the program starts.
  - The size of the memory is determined before the program starts.

  - (Example) int score_s[100];

  - If an input larger than the initially determined size comes in, it cannot be processed.
  - If a smaller input comes in, the remaining memory space is wasted.

Dynamic memory allocation

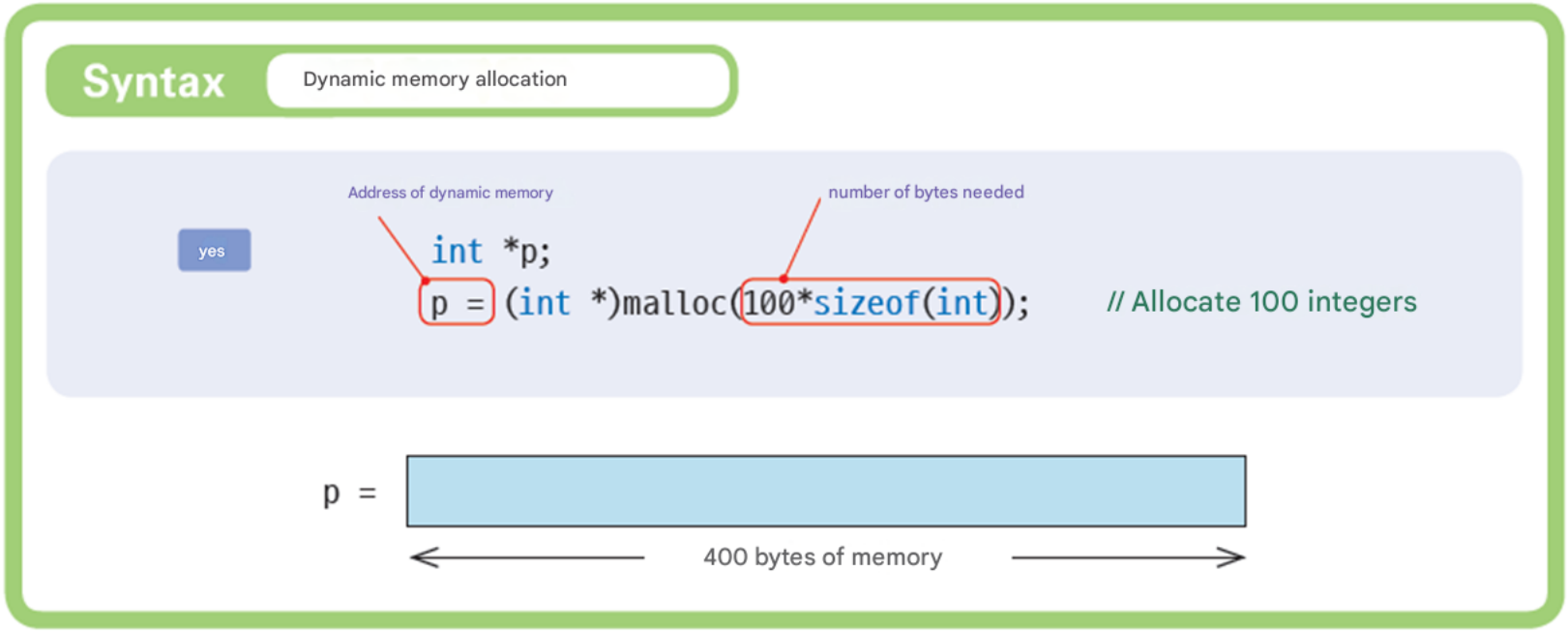
- **Dynamic memory allocation**
  - Dynamically allocating memory during execution
  - Return memory to the system when finished using it
  - score = (int *) malloc(100* sizeof (int));
  - Allocate only as much as you need and use memory very efficiently.




```
int main(void)
{
  int *score;
  score=(int)malloc(100*size(int));
  ...
}
```

program

Dynamic memory allocation

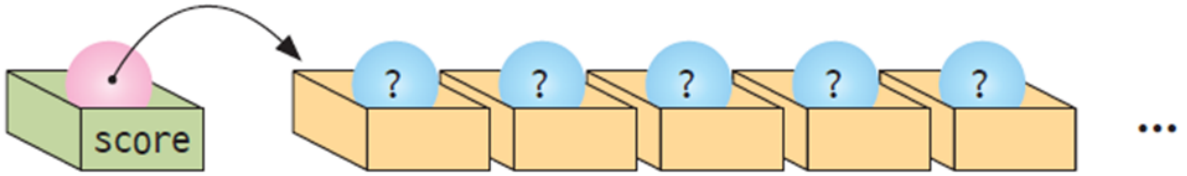


malloc() return value must be checked

```
int *score;
score = (int *)malloc(100*sizeof(int));

if( score == NULL ){
        ...  // error handling
}
```

convert to int pointer

Check if memory is allocated correctly



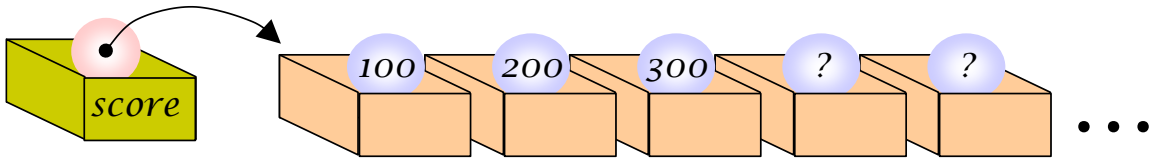
Dynamic Memory usage

How to use the allocated space?

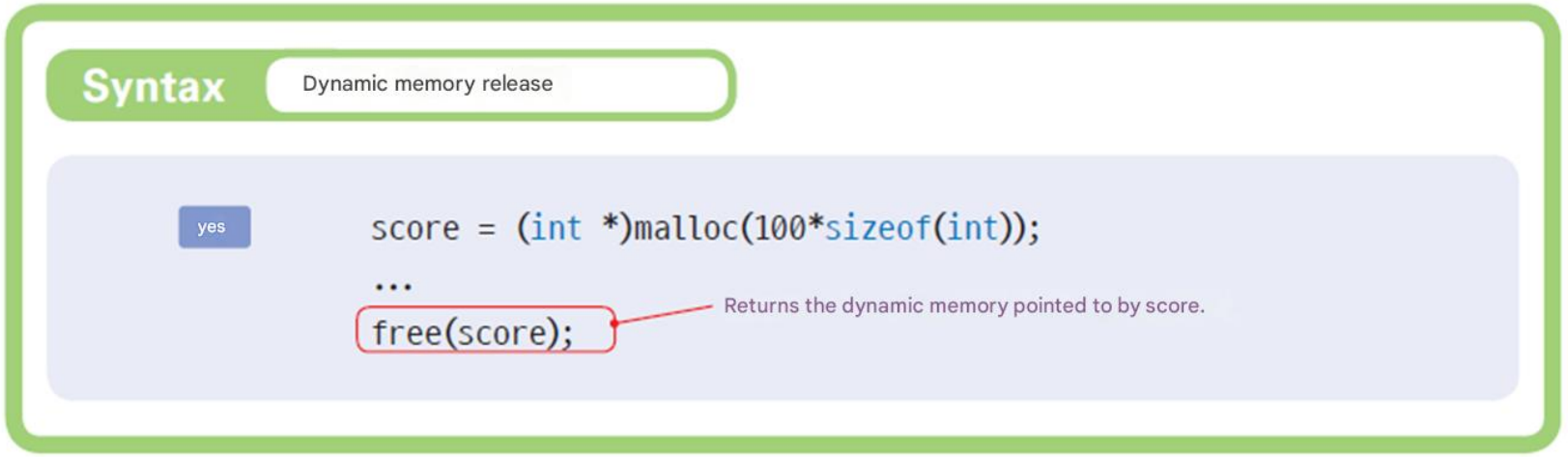
*Int *score = NULL;*

*score = (int *)malloc(100*sizeof(int));*

- First method : Using pointers
  - * score = 100;
  - * (score+1) = 200;
  - * (score+2) = 300;
  - …
- Second method : Treat dynamic memory like an array
  - score[0] = 100;
  - score[1] = 200;
  - score[2] = 300;
  - …



Dynamic memory return



Example

- Allocates space to store one integer, one real number, and one character, uses it, and then returns it .

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

int main( void )
{
        int *pi;
        double *pf;
        char *pc;

        pi = ( int *)malloc( sizeof ( int ));
        pf = (double *)malloc( sizeof (double));
        pc = (char *)malloc( sizeof (char));
        if (pi == NULL || pf == NULL || pc == NULL ) {
                // The return value is NULL cognition check
                printf ( " Dynamic memory allocation error \n" );
                exit(1);
        }
```

Example

```
        *pi = 100; // pi[0] = 100;
        *pf = 3.14; // pf[0] = 3.14;
        *pc = 'a'; // pc[0] = 'a';

        free(pi);
        free(pf);
        free(pc);
        return 0;
}
```

Example #2

- Let's create a dynamic memory that can store the integers 10, 20, and 30 .

```
#include < stdio.h >
#include < stdlib.h >
int main( void )
{
        int *list;
    list = ( int *) malloc (3 * sizeof ( int ));
        if (list == NULL ) { // The return value is NULL cognition check
                printf ( " Dynamic memory allocation error \n" );
                exit(1);
        }
        list[0] = 10;
        list[1] = 20;
        list[2] = 30;
    free(list);
         return 0;
}
```

Dynamic memory allocation

Dynamic memory freeing

Lab: Processing Using Dynamic Arrays

- Let's say you're writing a grade processing program. It asks the user how many students there are and allocates appropriate dynamic memory. It gets the grades from the user, stores them, and then prints them out .

```
Number of students : 3
Student #1 Grade : 10
Student #2 Grade : 20
Student #3 Grade : 30
Grade average = 20.00
```

Solution

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

int main( void )
{
        int *list;
        int i, students, sum=0;

        printf ( " Number of students : " );
        scanf ( "%d" , &students);

        list = ( int *)malloc(students * sizeof ( int ));
        if (list == NULL ) { // The return value is NULL cognition check
              printf ( " Dynamic memory allocation error \n" );
              exit(1);
        }
```

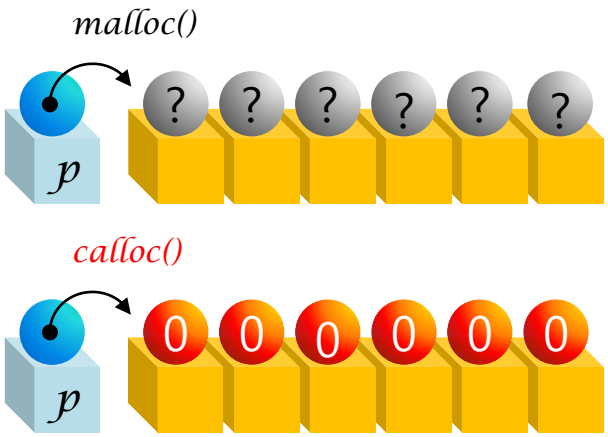
Solution

```
        for (i = 0; i<students; i++) {


                printf ( " Student #%d grade : " , i+1);
                scanf ( "%d" , &list[ i ]);
        }
        for (i = 0; i<students; i++)
                sum += list[ i ];

        printf ( " Grade average =%.2f \n" , (double)sum/students);
        free(list);
        return 0;
}
```

calloc()

- calloc() allocates memory initialized to 0.
- calloc() allocates memory on an item-by-item basis.
- (Example)

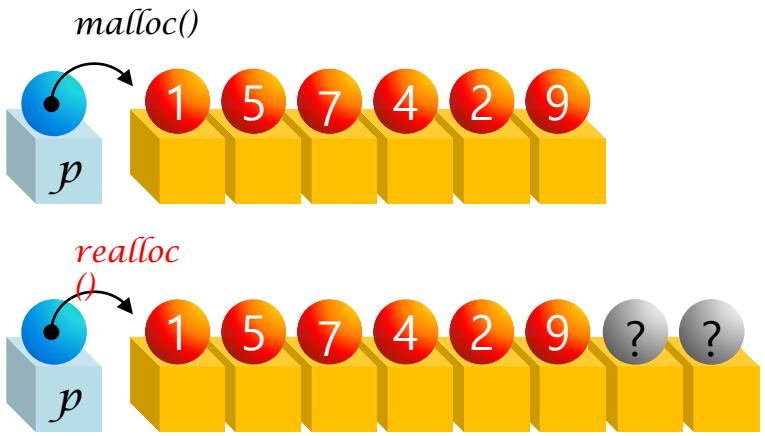
```
int *p;
p = (int *) calloc (5, sizeof (int));
```



realloc()

- The realloc() function changes the size of an allocated memory block.
- (Example)

```
int *p ;
p = (int *) malloc (5 * sizeof (int)));
p = realloc (p, 7 * sizeof (int)));
```



Tip

What happens if you don't return dynamic memory?

Dynamic memory is not released unless the programmer explicitly returns it. The operating system reserves a certain portion of memory as a heap and allocates dynamic memory from it. Therefore, dynamic memory has a fixed size, so if a certain program uses a lot of it, other programs will be restricted. In fact, if dynamic memory is not returned, the entire program gradually slows down.

The worst case scenario is to keep allocating and never returning anything. This can cause the operating system to crash. The code below is very wrong.

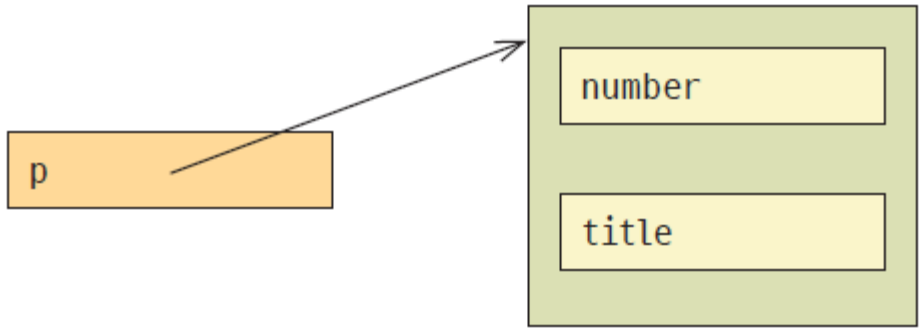
```
void sub()
{
    int *p;
    p = malloc( 100 * sizeof(int) );
    p = malloc( 100 * sizeof(int) );
    ...
    return;
}
```

The address to the previous memory block disappears.

Let's create a structure dynamically .

- Allocate space to store the structure and use it .

```
struct Book {
 int number;
 char title[50];
};

struct Book *p;
p = ( struct Book *)malloc(2 * sizeof ( struct Book));
```

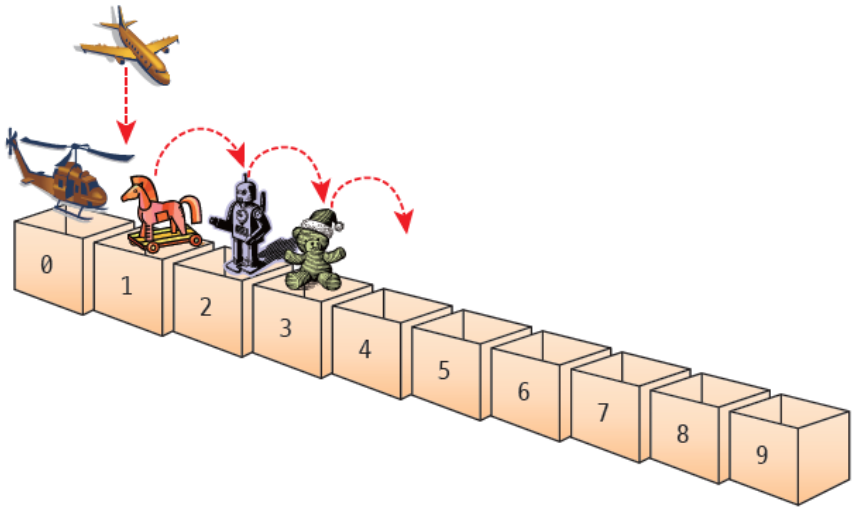


```c
#include < stdio.h >
#include < stdlib.h >
#include < string.h >
struct Book {
        int number;
        char title[50];
};

int main( void )
{
        struct Book *p;
        p = ( struct Book *) malloc (2 * sizeof ( struct Book ));
        if (p == NULL ) {
            printf ( " Memory allocation error \n" );
            exit(1);
        }
        p[0].number = 1;
        strcpy (p[0].title, "C Programming" );
        p[1].number = 2;
        strcpy (p[1].title, "Data Structure" );
        free(p);
        return 0;
}
```

Allocating an array of struc

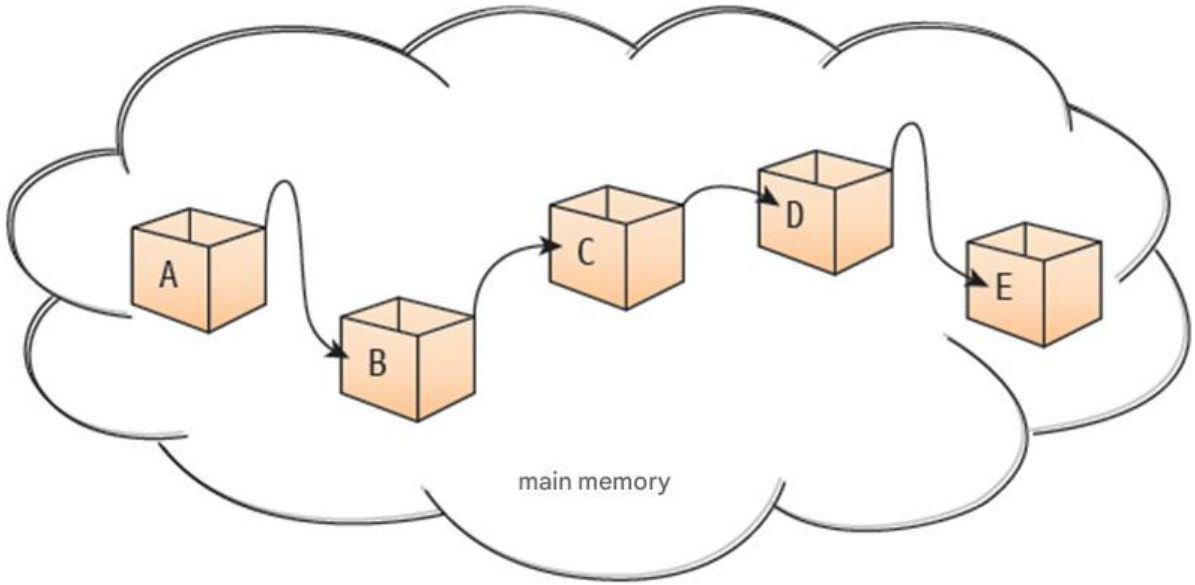
Array vs Linked List

- array
  - Pros : Simple and fast to implement.
  - Disadvantage : The size is fixed. It is difficult to insert or delete in the middle.



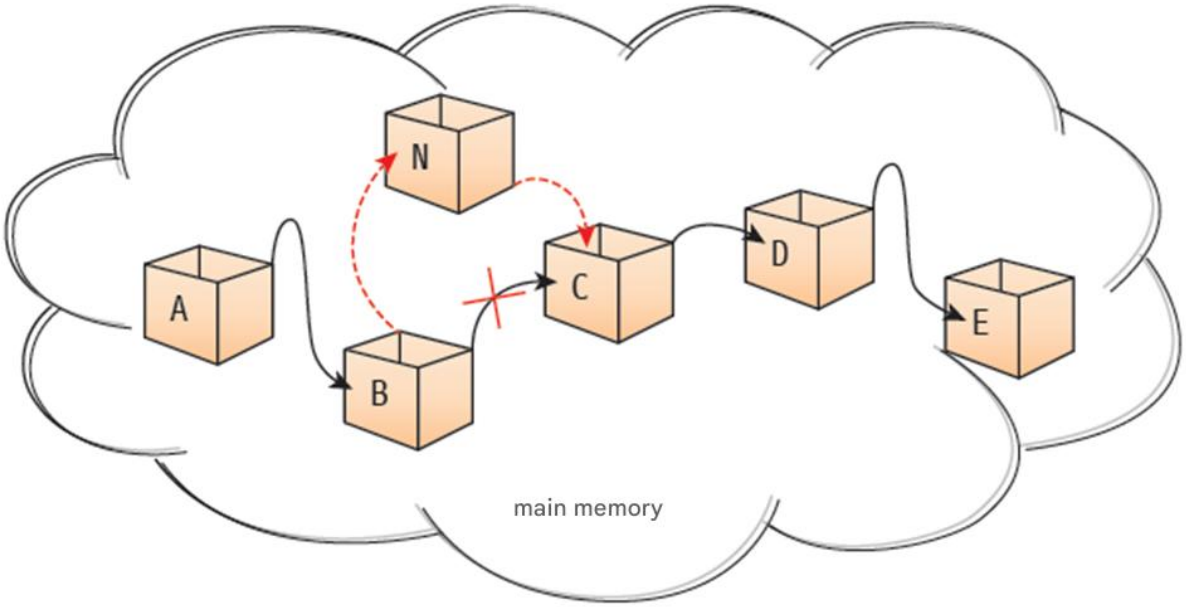
Array vs Linked List

- linked list
  - Each item uses a pointer to point to the next item .
  - Random access is difficult .
  - It is widely used to implement



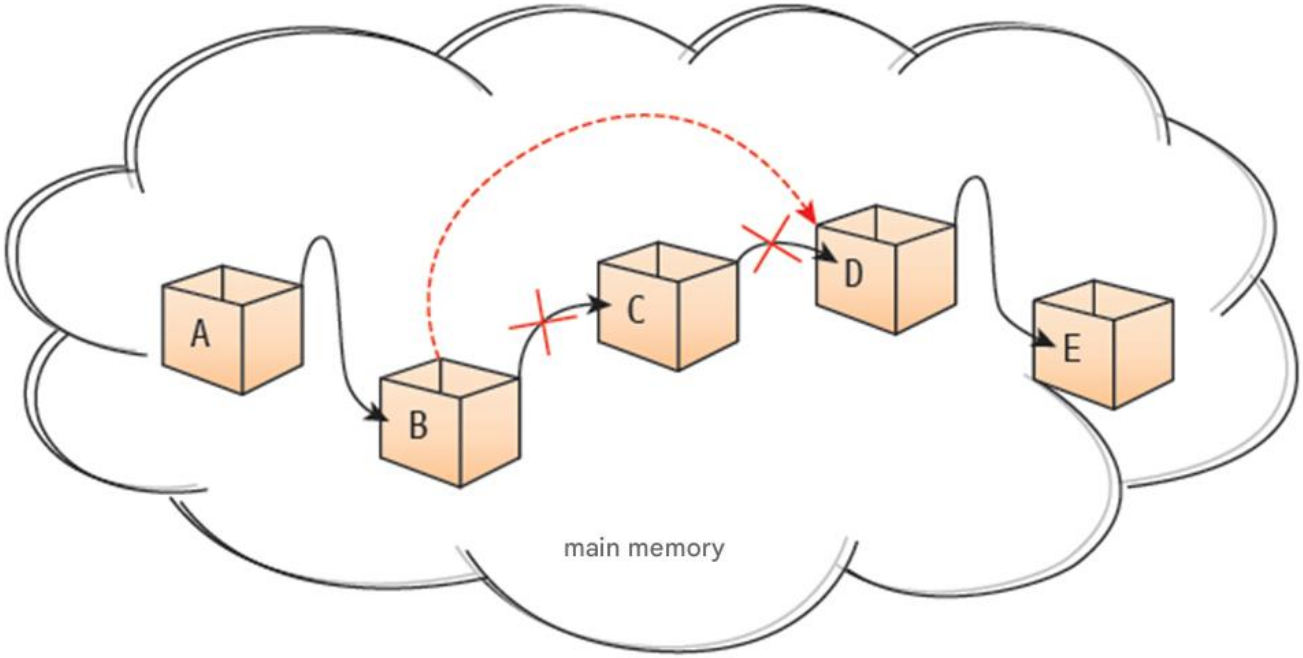
Insert operation in linked list

- or delete data in the middle .



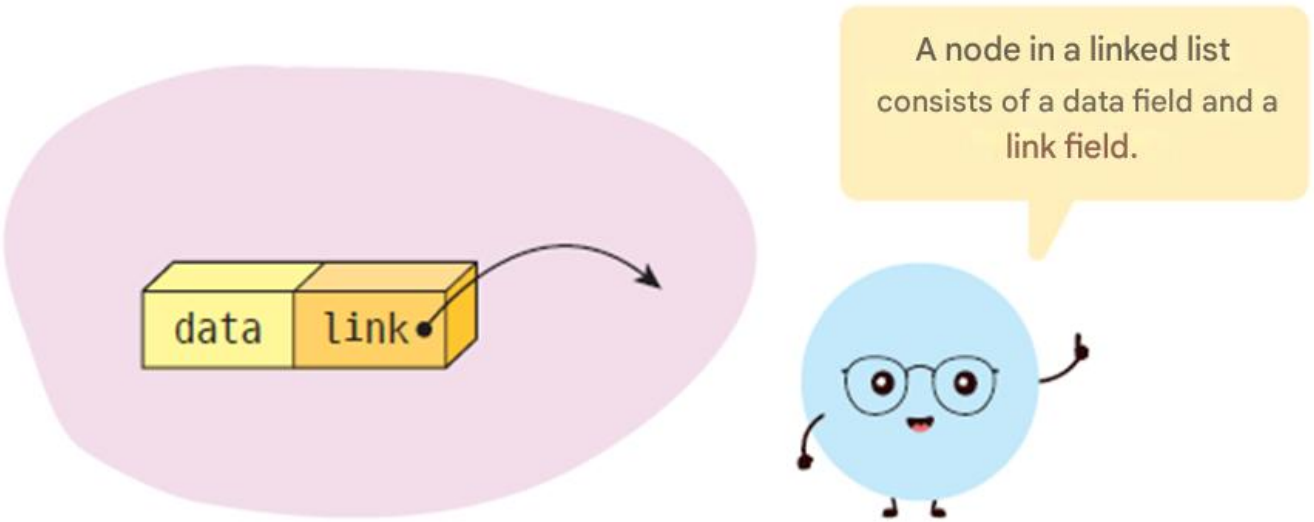
Delete operation in linked list

- or delete data in the middle .



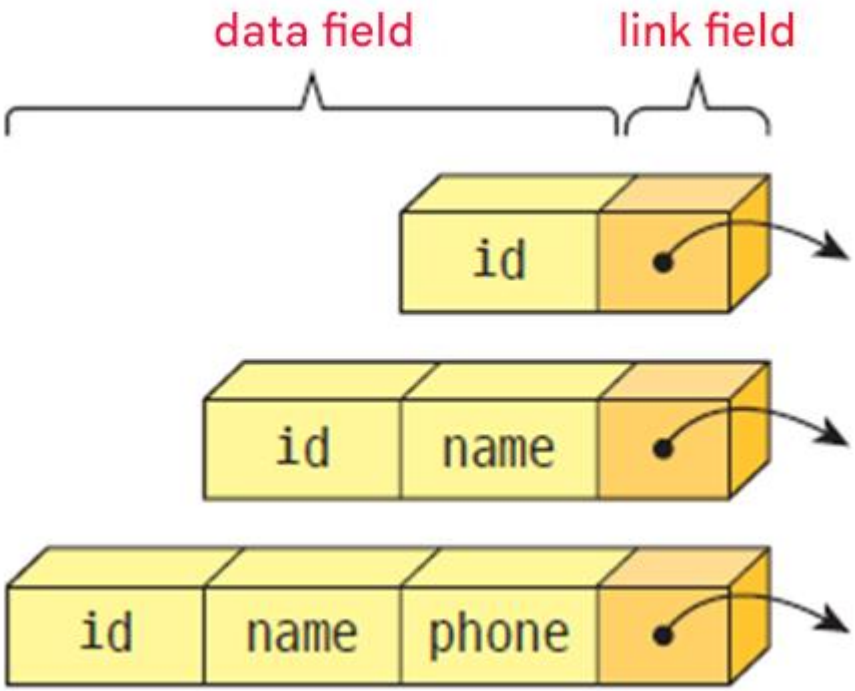
Structure of a linked list

- Node = data field + link field



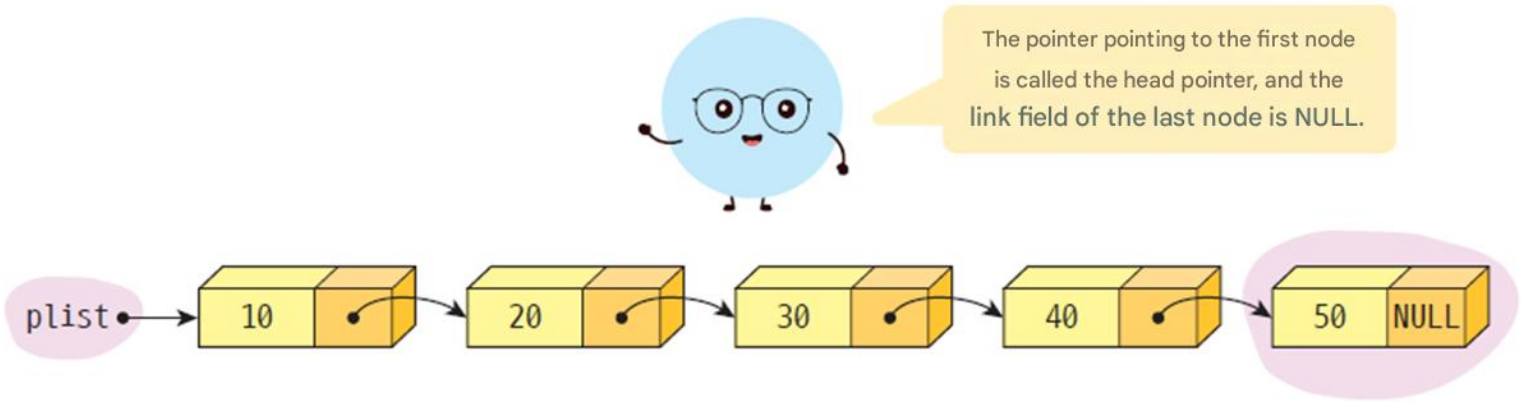
Data Fields

- The data field contains the data we want to store. The data can be an integer or it can be complex data such as a structure containing a student number, name, and phone number.



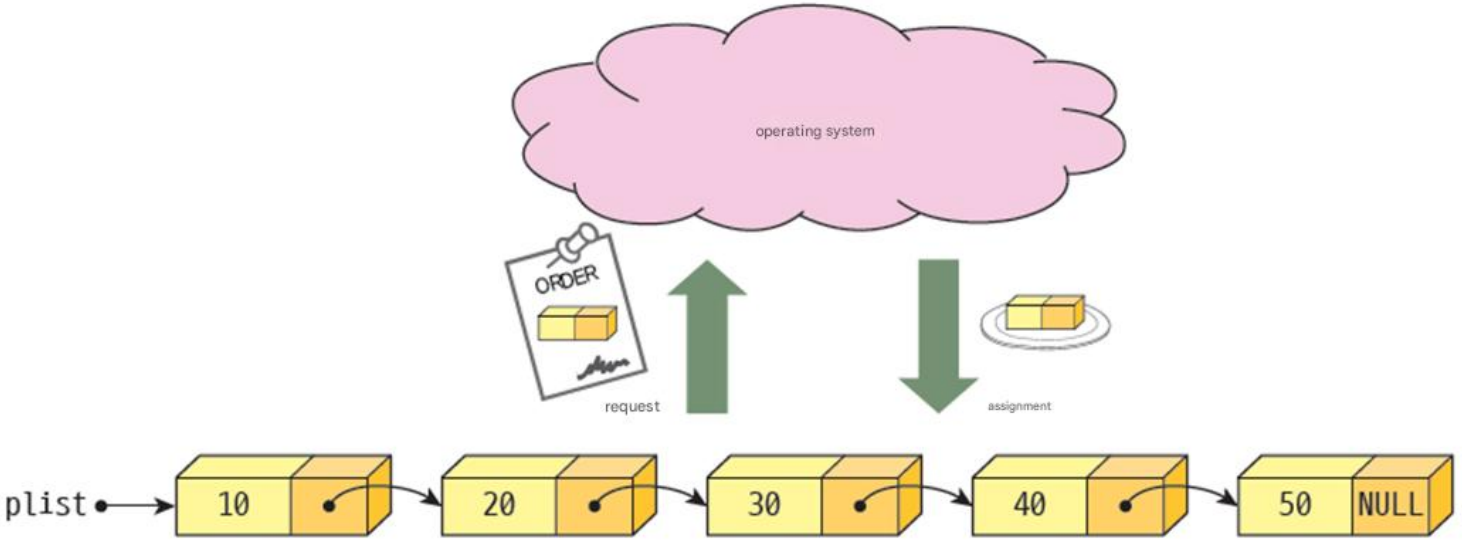
Structure of a linked list

- Head pointer : A pointer pointing to the first node.



Create a node

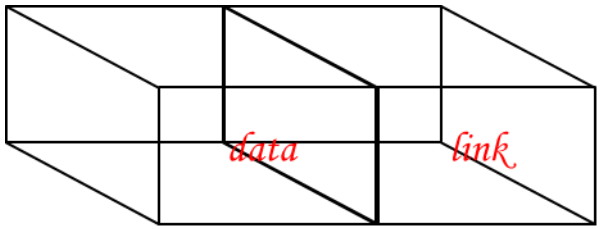
- Nodes are created dynamically .



Self-referential structure

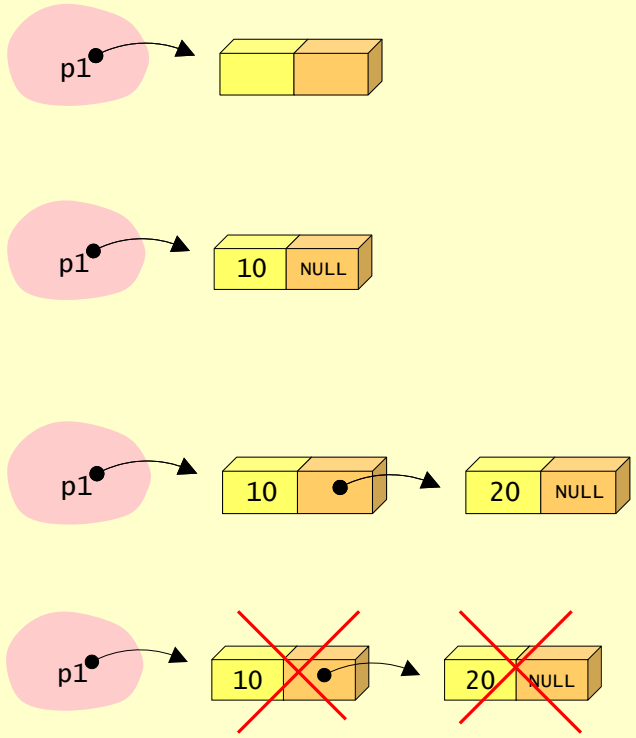
- **A self -referential structure** is a special structure that has a pointer to a structure of the same type among its members.

```
typedef  struct NODE {
int data;
 struct NODE *link;
} NODE;
```



Creating a simple linked list

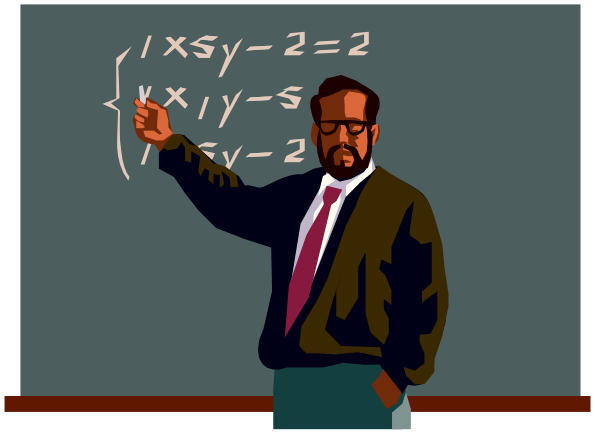
```
NODE* p1;
p1 = (NODE*)malloc( sizeof (NODE));

p1->data = 10;
p1->link = NULL ;


NODE* p2;
p2 = (NODE*)malloc( sizeof (NODE));
p2->data = 20;
p2->link = NULL ;
p1->link = p2;

free( p1 );
free( p2 );
```



Manual memory management vs automatic memory management

- C language presents several problems.
  - You can forget to free memory. If you don't free memory after you're done using it, you can end up with a memory leak.
  - It may free memory too early : meaning the program releases memory while it is still being used. If another part of the program later tries to access that freed memory, the program can crash because the data no longer exists.
- Because this problem was undesirable, modern languages added automatic memory management : a garbage collector took care of it.
- Automatic memory management offers many benefits to programmers. However, automatic memory management comes at a cost.
  In many programming languages with automatic memory management, all execution stops while the garbage collector searches for and deletes objects to collect.
- However, for long-running applications where performance is critical, manual memory management is still used. The most representative languages are the *C and C++ languages* that we are learning .

Q & A




5 4 3 2 1
The
END


THANK
YOU