A linked list is a *data structure* that uses a "chain" of node objects, connected by pointers, to organize a collection of user data values.

Here's a fairly typical conceptual view of a doubly-linked list:

```
Head node

12

41

5

19

27

17

23

8

Tail node

33

32
```
A linked list implementation will typically provide at least:

- initialization function to set up basic structure for an empty list
- insert functions to add new element to the list; at front, at rear, at user-selected position, ordered insertion
- remove function to remove element from the list
- find function to determine whether a given element occurs in the list
- clear function to restore the list to an empty state

In C we would organize this as a pair of `struct` types (list and node) and a collection of associated functions.
Three Views

There are three general approaches to making a linked list in C:

- hard-code a data type into the node, requiring the creation of near-duplicate implementations in order to create lists holding data of different types

- use a `void*` member in the node which can then point to a data object of any type, making the list versatile but losing all ability to do compile-time type checking

- embed the node type as a member within the user data object, which maximizes flexibility and preserves compile-time type checking, but requires the user to modify the data type or to create a new "bundling" type
A Payload Type

```c
#ifndef POINT_H
#define POINT_H

struct _Point {
    int X;
    int Y;
};

typedef struct _Point Point;

void Point_Init(Point* const pPt, int X, int Y);

#endif
```

We'll use this in some examples.
A Payload-specific Node Type

```c
struct _DNode {
    Point Payload;          // hard-coded user data type
    struct _DNode* Prev;     // pointer to preceding node in list
    struct _DNode* Next;     // pointer to succeeding node in list
};

typedef struct _DNode DNode;
```

This can be the basis for a list implementation that can store an arbitrary number of Point variables.
A Generic Node Type

```c
#ifndef DNODE_H
#define DNODE_H

struct _DNode {
    void* Payload;          // generic pointer to user data object
    struct _DNode* Prev;
    struct _DNode* Next;
};

typedef struct _DNode DNode;
#endif
```

The `void*` member can be used to point to `Point` variables, or anything else.
Wrapping the Node in the Payload

```c
#ifndef DLIST_H
#define DLIST_H

// List node:
struct _DNode {
    struct _DNode *prev;  // Previous list element
    struct _DNode *next;  // Next list element
};

// List object:
struct _DList {
    struct _DNode head;   // List head
    struct _DNode tail;   // List tail
};

typedef struct _DNode DNode;
typedef struct _DList DList;
#endif
```
Wrapping the Node in the Payload

```c
#ifndef POINTLISTELEM_H
#define POINTLISTELEM_H

#include "Point.h"
#include "DList.h"

struct _PointListElem {
    Point Pt;
    DNode node;
};

typedef struct _PointListElem PointListElem;

void PointListElem_Init(PointListElem* const pLE, const Point* const Pt);

#endif
```

This allows us use a single DList of DNode objects with any user data type.
Example: DNode type

We'll develop an example using the payload-specific view:

```c
struct _DNode {
    Point Payload;

    struct _DNode* Prev;
    struct _DNode* Next;
};

typedef struct _DNode DNode;

void DNode_init(DNode* const pNode, const Point* const pPayload);
```
The initialization function plays much the same role as a constructor in OOP languages:

```c
void DNode_init(DNode* const pNode, const Point* const pPayload) {
    assert(pNode != NULL); // terminate pgm if pNode is NULL
    assert(pPayload != NULL); // or if pPayload is NULL
    pNode->Prev = NULL; // set both ptrs to NULL
    pNode->Next = NULL;
    pNode->Payload = *pPayload; // install payload in node
}
```

The use of `assert()` is somewhat analogous to throwing an exception in OOP. If the Boolean parameter to `assert()` is false, the program will be terminated with an error message.

It's an extreme response to a logical error; an alternative would be to use a `bool` return type and return `false` if the parameters fail the NULL tests.
Example: Dnode "constructor"

```c
void DNode_init(DNode* const pNode, const Point* const pPayload) {
    assert(pNode != NULL); // terminate pgm if pNode is NULL
    pNode->Prev = NULL;    // set both ptrs to NULL
    pNode->Next = NULL;
    pNode->Payload = *pPayload; // install payload in node
}
```

A newly-created node has no neighbors, so it's necessary to set the pointer fields to NULL. One of the most common errors in list implementations is to fail to set pointers to NULL when they do not have a target.
Example: Dnode "constructor"

```c
void DNode_init(DNode* const pNode, const Point* const pPayload) {
    assert(pNode != NULL); // terminate pgm if pNode is NULL
    pNode->Prev = NULL; // set both ptrs to NULL
    pNode->Next = NULL;
    pNode->Payload = *pPayload; // install payload in node
}
```

Note the uses of const in the parameter list:

- pNode's target (the node itself) must be modified, but pNode should not, even though it's a copy of the caller's pointer
- The target of pPayload will be copied into the field Payload; a copy of the target is made when the assignment is carried out.
- The payload is passed into the initializer by pointer in order to avoid the cost of making a copy of it when the initializer is called; if we passed it by value it would then be copied twice.
Example: DList type

```c
struct _DList {  // The list object holds two dummy nodes;  
    // that is, they do not hold data.
    DNode Head;  // Head points to the first node in the list;
    DNode Tail;  // Tail points to the last node in the list.
};

typedef struct _DList DList;

void DList_init(DList* const pList);
bool DList_isempty(const DList* const pList);

bool DList_push_front(DList* const pList, const Point* const pPt);
bool DList_push_back(DList* const pList, const Point* const pPt);

const Point* DList_find(const DList* const pList,
                        const Point* const pPt);

bool DList_remove(DList* const pList, const Point* const pPt);
void DList_clear(DList* const pList);
```
Example: DList "constructor"

```c
void DList_init(DList* const pList) {

    assert(pList != NULL);

    // In an empty Dlist, Head's successor is Tail:
    pList->Head.Prev = &pList->Head;   // 1
    pList->Head.Next = &pList->Tail;   // 2

    // ... and Tail's predecessor is Head:
    pList->Tail.Next = &pList->Tail;   // 3
    pList->Tail.Prev = &pList->Head;   // 4
}
```
Example: DList_isempty()

```c
bool DList_isempty(const DList* const pList) {
    assert(pList != NULL);
    return (pList->Head.Next == &pList->Tail);
}
```

The expression in the `return` statement is rather interesting:

```c
pList->Head.Next == &pList->Tail
```

- address of successor to Head
- address of Tail

The expression `&pList->Tail` is equivalent to `&(pList->Tail)`.

Note that the logic of this function is tied to the configuration the initialization function creates for an empty list.
Example: DList_push_front()

```c
bool DList_push_front(DList* const pList, const Point* const pPt) {
    assert(pList != NULL);
    assert(pPt != NULL);

    DNode* newNode = malloc(sizeof(DNode)); // try to make new node
    if (newNode == NULL) return false;

    DNode_init(newNode, pPt); // initialize new node

    newNode->Next = pList->Head.Next; // make Head's successor the
    // new node's successor
    newNode->Prev = &pList->Head; // make Head the new node's
    // predecessor
    pList->Head.Next->Prev = newNode; // make new node the
    // predecessor of Head's
    // current successor
    pList->Head.Next = newNode; // make new node Head's
    // successor

    return true;
}
```
Example: DList\_push\_front()

```c
bool DList\_push\_front(DList* const pList, const Point* const pPt) {
    . . .
    newNode->Next = pList->Head.Next;    // 1
    newNode->Prev = &pList->Head;        // 2
    pList->Head.Next->Prev = newNode;    // 3
    pList->Head.Next = newNode;          // 4
    . . .                                 // successor
}
```
Example: DList_find()

```c
const Point* DList_find(const DList* const pList,
                        const Point* const pPt) {

    assert(pList != NULL);

    DNode* curr = pList->Head.Next; // get ptr to first node

    while ( curr != &pList->Tail ) { // quit at Tail

        if ( Point_equals(pPt, &curr->Payload) ) // match?
            return &curr->Payload;

        curr = curr->Next; // step to next node
    }

    return NULL;
}
```

The function returns a `const` pointer to the matching data object (or `NULL`).

So, the user cannot use this access to modify the data object within the list.