Lists in C

Personal Software Engineering
But First - How Much Space Is Needed?

For strings, we can use strlen:

```
char *p_copy = malloc( strlen("Hello")+1 ) ;
```

But what about other types (ints, doubles, structs, etc.)?

This is the purpose of the `sizeof` operator!
**sizeof** for basic types

\[ \text{sizeof}(\text{type}) = \text{#bytes needed to hold a type value} \]

\[ \text{sizeof}(\text{variable}) = \text{#bytes needed to hold variable's type.} \]

Examples (current 32 and 64 bit systems):

\[ \text{sizeof}(\text{char}) = 1 \]
\[ \text{sizeof}(\text{short}) = 2 \]
\[ \text{sizeof}(\text{int}) = 4 \]
\[ \text{sizeof}(\text{float}) = 4 \]
\[ \text{sizeof}(\text{long}) = 8 \]
\[ \text{sizeof}(\text{double}) = 8 \]
\[ \text{sizeof}(\text{char }*) = 4 \text{ (32-bit systems) / 8 (64-bit systems)} \]

**NOTE:** all pointers to any type have the same size!
### `sizeof` for array types

```c
double sampledData[100];
sizeof(sampledData); // = 100 * 8 = 800

char string[81];
sizeof(string); // = 81 * 1 = 81

BUT
void foo(char buffer[81]) { . . . }  
sizeof(buffer); // = 8 !!

WHY?
Because array arguments are **really** pointers!
The function header above is equivalent to:

```c
void foo(char *buffer) { . . . }
```
sizeof for structs

typedef struct _node {
  int contents ;
  struct _node *next ;
} node ;

sizeof(node) == # bytes required to hold the structure.
  == sizeof(int) + size(node *) + padding

Padding is needed to assure data are aligned on the proper boundary:
  ints on 4 byte boundaries
  shorts on 2 byte boundaries
  doubles and pointers on 8 byte boundaries

Padding is dictated by the way CPU's access memory.
Singly Linked Lists

A **(singly) linked list** comprises a set of **nodes**, each node having a **pointer** to the next node in the list. We keep a pointer to the first node in a **list head pointer**.

Since lists can grow and shrink dynamically, space for the list nodes is allocated and released dynamically using **malloc** and **free**.
typedef struct _node {
    int contents;
    struct _node *next;
} node;

define p_head = NULL;
define np = malloc( sizeof(node) ) ; np->contents = 800 ;
np->next = p_head ; p_head = np ;
np = malloc( sizeof(node) ) ; np->contents = 150 ;
np->next = p_head ; p_head = np ;
np = malloc( sizeof(node) ) ; np->contents = 100 ;
np->next = p_head ; p_head = np ;
Linked List Example in C

typedef struct _node {
    int contents ;
    struct _node *next ;
} node ;

node *p_head = NULL ;
node *np = malloc( sizeof(node) ) ; np->contents = 800 ;
np->next = p_head ; p_head = np ;
np = malloc( sizeof(node) ) ; np->contents = 150 ;
np->next = p_head ; p_head = np ;
np = malloc( sizeof(node) ) ; np->contents = 100 ;
np->next = p_head ; p_head = np ;

Definition of the node type with a field to hold information (contents) and a pointer to the next node. NULL will mark the list end.
typedef struct _node {
    int contents;
    struct _node *next;
} node;

node *p_head = NULL;

node *np = malloc(sizeof(node)); np->contents = 800;
np->next = p_head; p_head = np;
np = malloc(sizeof(node)); np->contents = 150;
np->next = p_head; p_head = np;
np = malloc(sizeof(node)); np->contents = 100;
np->next = p_head; p_head = np;

p_head = NULL for the initial (empty) list.
Linked List Example in C

typedef struct _node {
    int contents;
    struct _node *next;
} node;

node *p_head = NULL;
node *np = malloc(sizeof(node)) ; np->contents = 800 ;
np->next = p_head ; p_head = np ;
np = malloc(sizeof(node)) ; np->contents = 150 ;
np->next = p_head ; p_head = np ;
np = malloc(sizeof(node)) ; np->contents = 100 ;
np->next = p_head ; p_head = np ;
typedef struct _node {
    int contents;
    struct _node *next;
} node;

node *p_head = NULL;
node *np = malloc(sizeof(node)); np->contents = 800;
np->next = p_head; p_head = np;
np = malloc(sizeof(node)); np->contents = 150;
np->next = p_head; p_head = np;
np = malloc(sizeof(node)); np->contents = 100;
np->next = p_head; p_head = np;

np's next is copied from p_head
p_head is set to np
typedef struct _node {
    int contents;
    struct _node *next;
} node;

node *p_head = NULL;
node *np = malloc(sizeof(node)); np->contents = 800;
np->next = p_head; p_head = np;
np = malloc(sizeof(node)); np->contents = 150;
np->next = p_head; p_head = np;
np = malloc(sizeof(node)); np->contents = 100;
np->next = p_head; p_head = np;

Allocate space for a node and assign the address to np
Set the contents to 150
```c
typedef struct _node {
    int contents;
    struct _node *next;
} node;

node *p_head = NULL;
node *np = malloc(sizeof(node)); np->contents = 800;
np->next = p_head; p_head = np;
np = malloc(sizeof(node)); np->contents = 150;
np->next = p_head; p_head = np;
np = malloc(sizeof(node)); np->contents = 100;
np->next = p_head; p_head = np;
```

np's next is copied from p_head
p_head is set to np
typedef struct _node {
    int contents ;
    struct _node *next ;
} node ;

node *p_head = NULL ;
node *np = malloc( sizeof(node) ) ; np->contents = 800 ; np->next = p_head ; p_head = np ;
np = malloc( sizeof(node) ) ; np->contents = 150 ; np->next = p_head ; p_head = np ;
np = malloc( sizeof(node) ) ; np->contents = 100 ; np->next = p_head ; p_head = np ;

Allocate space for a node and assign the address to np
Set the contents to 100
typedef struct _node {
  int contents;
  struct _node *next;
} node;

node *p_head = NULL;
node *np = malloc(sizeof(node)); np->contents = 800;
np->next = p_head; p_head = np;
np = malloc(sizeof(node)); np->contents = 150;
np->next = p_head; p_head = np;
np = malloc(sizeof(node)); np->contents = 100;
np->next = p_head; p_head = np;

np's next is copied from p_head
p_head is set to np
Linked List Example in C

- Some interesting questions:
  - How can we find the length of a list?
  - How can we add a node with the value 999 to the end of the list (rather than the head)?
  - How can we add a node with a new value (say 777) before the node at a given position (say 1)?
  - How can we the position of a node with a desired value?
  - How can we remove a node from the list?