

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

sizeof(*type*) = #bytes needed to hold a type value

sizeof(*variable*) = #bytes needed to hold variable's type.

Examples (current 32 and 64 bit systems):

sizeof(char) = 1

sizeof(short) = 2

sizeof(int) = 4

sizeof(float) = 4

sizeof(long) = 8

sizeof(double) = 8

sizeof(char *) = 4 (32-bit systems) / 8 (64-bit systems)

NOTE: all pointers to any type have the same size!

sizeof for array types

```
double sampledata[100] ;  
sizeof(sampledata) ;           // = 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 above is equivalent to:

```
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.

What are lists?

An array holds multiple items in memory

- And they are all sequential (in memory)

A list also holds multiple items in memory

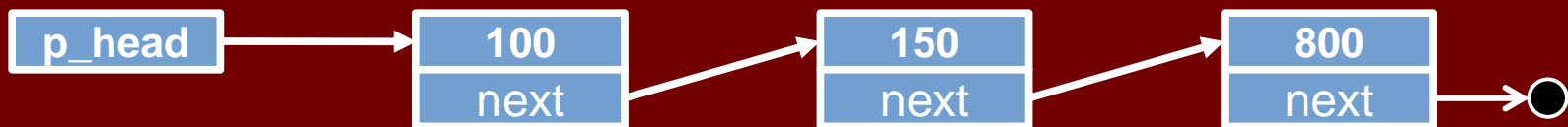
- But each item is in a different (unknown) location in memory!

Lists are therefore much more useful (flexible) than arrays, because you don't need large amounts of sequential memory to hold all the items

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**.

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 ;
```

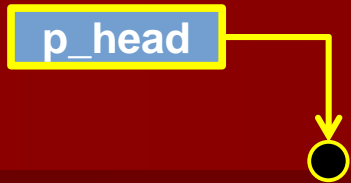

Linked List Example in C

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

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.

```
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 ;
```

Linked List Example in C



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

p_head = NULL for the
initial (empty) list.

```
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 ;
```

Linked List Example in C

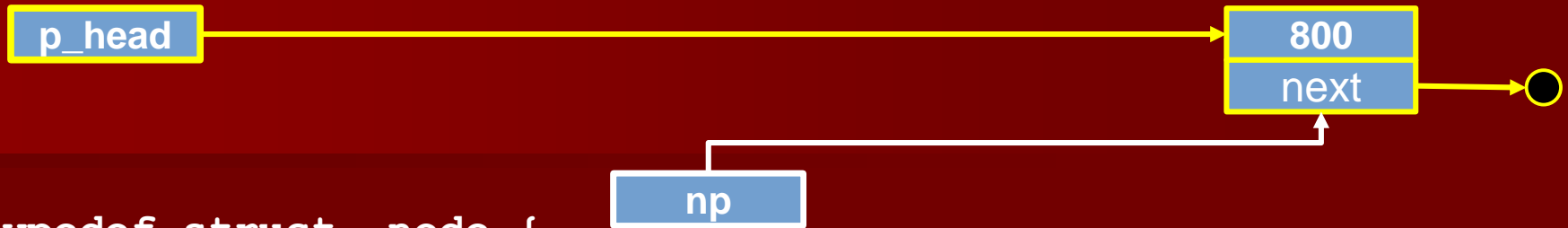


```
typedef struct _node {  
    int contents ;  
    struct _node *next ;  
} node ;
```

```
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 ;
```

Allocate space for a node and
assign the address to np
Set the contents to 800

Linked List Example in C

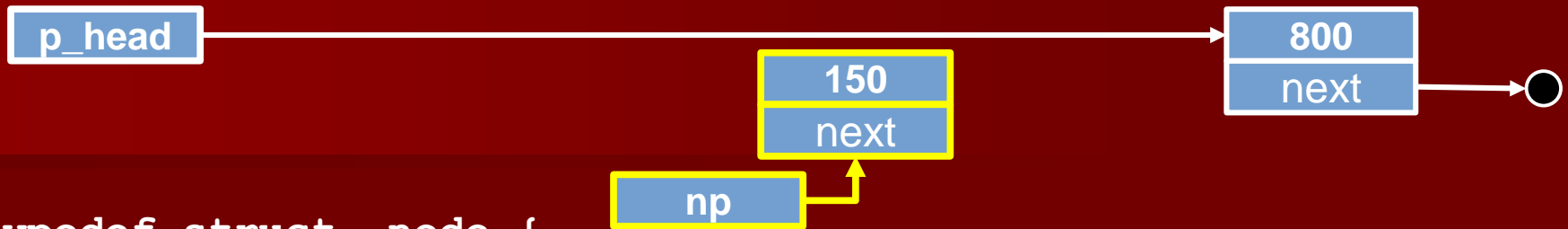


```
typedef struct _node {  
    int contents ;  
    struct _node *next ;  
} node ;
```

```
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 ;
```

np's next is copied from p_head
p_head is set to np

Linked List Example in C

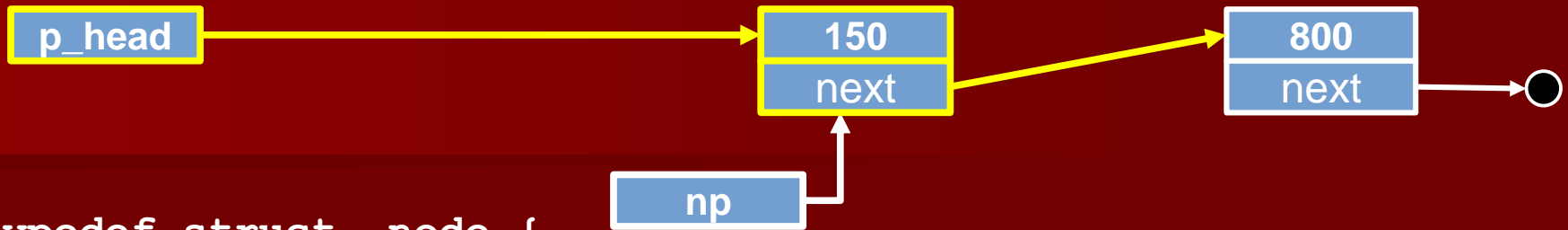


```
typedef struct _node {
    int contents ;
    struct _node *next ;
} node ;
```

```
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 ;
```

Allocate space for a node and
assign the address to np
Set the contents to 150

Linked List Example in C

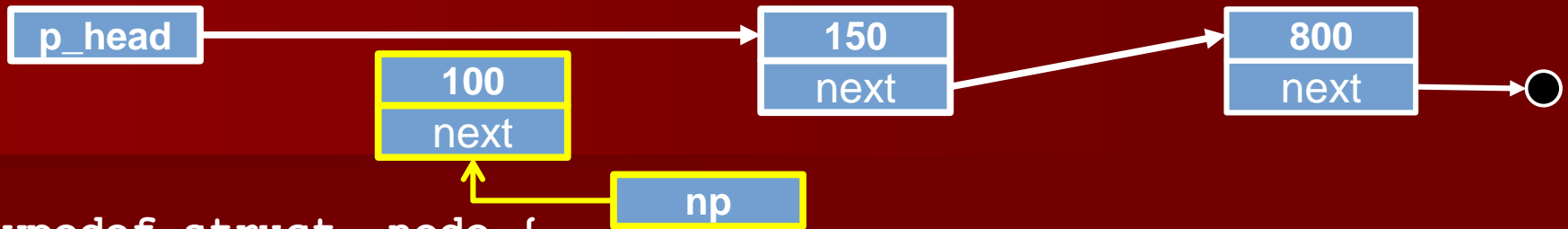


```
typedef struct _node {  
    int contents ;  
    struct _node *next ;  
} node ;
```

```
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 ;
```

np's next is copied from p_head
p_head is set to np

Linked List Example in C

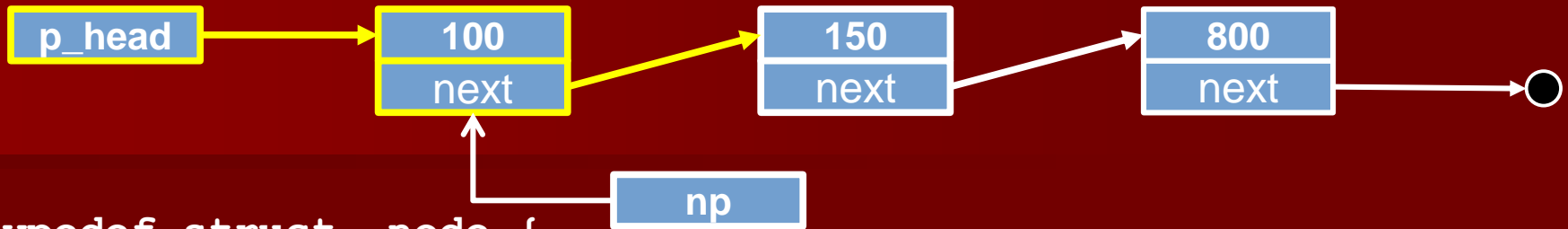


```
typedef struct _node {
    int contents ;
    struct _node *next ;
} node ;
```

```
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 ;
```

Allocate space for a node and
assign the address to np
Set the contents to 100

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 ;
```

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 find the position of a node with a desired value?
 - How can we remove a node from the list?