CSE 31
Computer Organization

Lecture 7 – C structs, Final Points about C and Memory Management in C

Announcements

- Labs
  - Lab 2 due this week (**with 7 days grace period** after due date)
    - » Demo is REQUIRED to receive full credit
  - Lab 3 out this week
    - » Due at 11:59pm on the same day of your next lab (with 7 days grace period after due date)
    - » You must demo your submission to your TA within 14 days from posting of lab
    - » Demo is REQUIRED to receive full credit
- Reading assignments
  - Chapter 4-6 of K&R (C book) to review C/C++ programming
  - Reading 01 (zyBooks 1.1 – 1.5) due 13-FEB
    - » Complete **Participation Activities** in each section to receive grade towards Participation
    - » IMPORTANT: Make sure to submit score to CatCourses by using the link provided on CatCourses
- Homework assignment
  - Homework 01 (zyBooks 1.1 – 1.5) due 20-FEB
    - » Complete **Challenge Activities** in each section to receive grade towards Homework
    - » IMPORTANT: Make sure to submit score to CatCourses by using the link provided on CatCourses

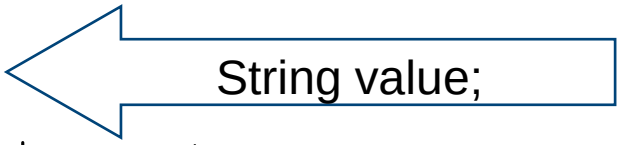
Announcements

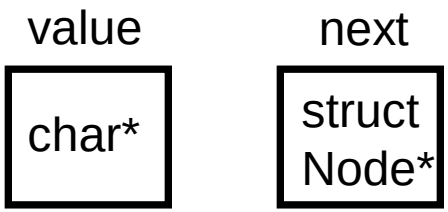
- Project 01
  - Due 17-MAR
  - Can work in teams of 2 students
    - » Each team member must identify teammate in “Comments…” text-box at the submission page
    - » If working in teams, each student must submit code (can be the same as teammate) and demo individually
    - » Grade can vary among teammates depending on demo
  - Demo required for project grade
    - » No partial credit for submission without demo
  - No grace period
    - » Must complete submission and demo by due date.

Linked List Example

- Let's look at an example of using structures, pointers, `malloc()`, and `free()` to implement a linked list of strings.

```
/* node structure for linked list */
struct Node {
    char *value;
    struct Node *next;
};
```





How big are structs?

- Recall C operator `sizeof()` which gives size in bytes (of type or variable)

- How big is `sizeof(p)`?

```
struct p {
    char x;
    int y;
};
```

  - 5 bytes? 8 bytes?
  - Compiler may word align integer `y`
    - » See https://en.wikipedia.org/wiki/Data_structure_alignment
  - More on this later lectures

typedef simplifies the code

```
struct Node {
    char *value;
    struct Node *next;
  };

/* "typedef" means define a new type */
typedef struct Node NodeStruct;
```

... OR ...

```
typedef struct Node {
    char *value;
    struct Node *next;
  } NodeStruct;
```

... THEN

```
typedef NodeStruct * List;
typedef char * String;
```

```
/* Note similarity! */
/* C++ */
/* To define 2 nodes */

struct Node {
    char *value;
    struct Node *next;
} node1, node2;
```

Linked List Example

```
/* Add a string to an existing list */
List cons(String s, List list)
{
  List node = (List) malloc(sizeof(NodeStruct));

  node->value = (String) malloc (strlen(s) + 1);
  strcpy(node->value, s);
  node->next = list;
  return node;
}
```

List is a NodeStruct pointer type

String is a char pointer type

```
typedef struct Node {
  char *value;
  struct Node *next;
} NodeStruct;
```

… THEN

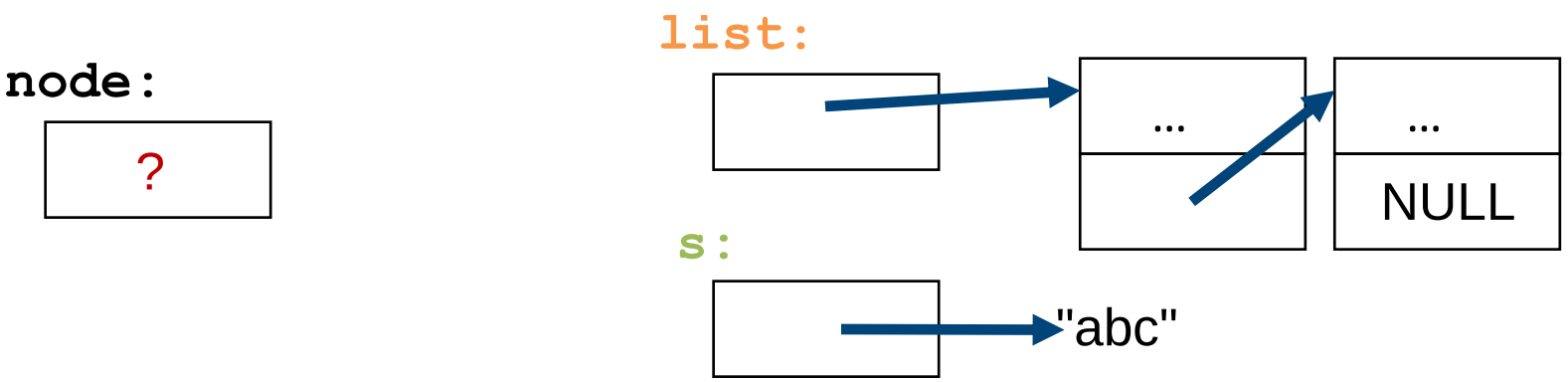
```
typedef NodeStruct * List;
typedef char * String;
```

- Somewhere in calling function…

```
String s1 = "abc", s2 = "cde";
List theList = NULL;
theList = cons(s2, theList);
theList = cons(s1, theList);
/* or embedded
theList = cons(s1, cons(s2, NULL)); */
```

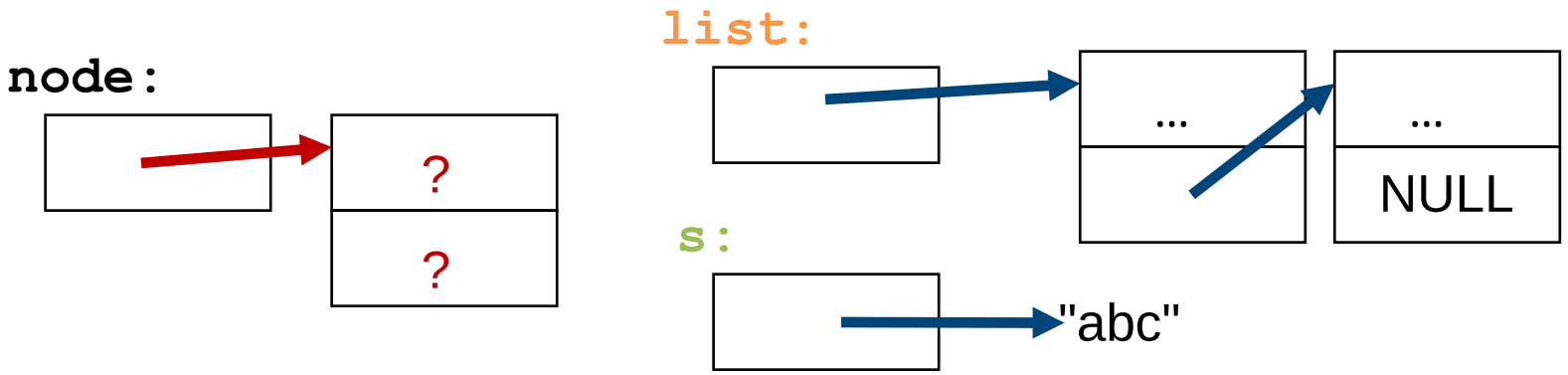
Linked List Example

```
/* Add a string to an existing list */
List cons(String s, List list)
{
  List node = (List) malloc(sizeof(NodeStruct));

  node->value = (String) malloc (strlen(s) + 1);
  strcpy(node->value, s);
  node->next = list;
  return node;
}
```



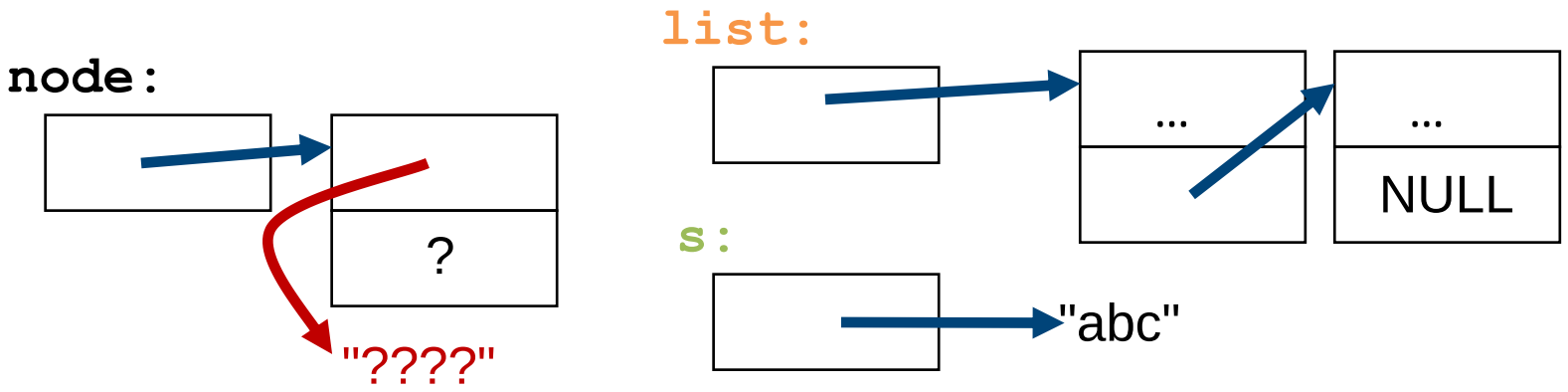
Linked List Example

```
/* Add a string to an existing list */
List cons(String s, List list)
{
  List node = (List) malloc(sizeof(NodeStruct));

  node->value = (String) malloc (strlen(s) + 1);
  strcpy(node->value, s);
  node->next = list;
  return node;
}
```



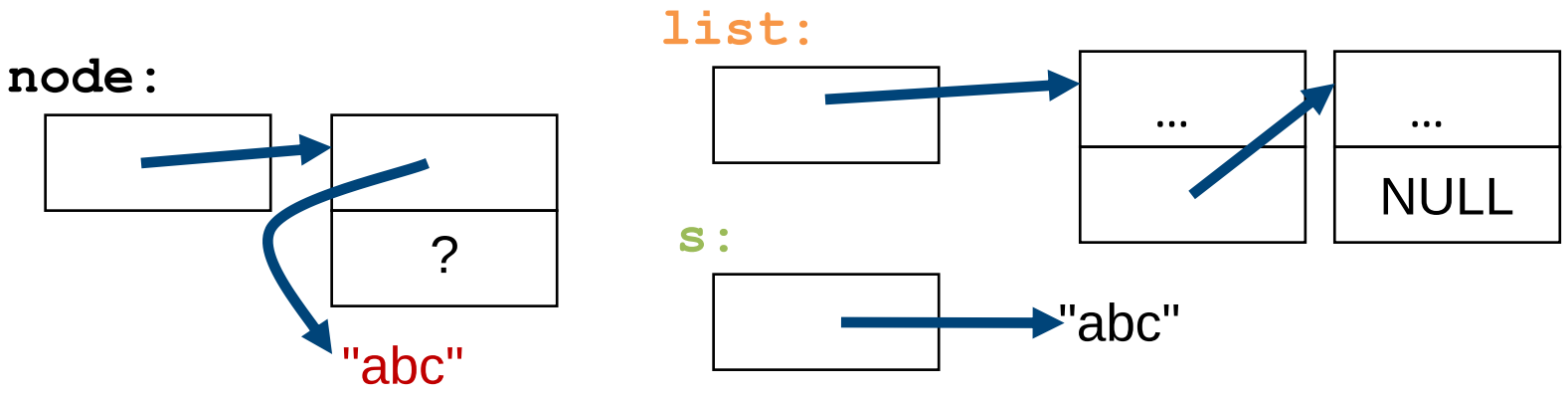
Linked List Example

```
/* Add a string to an existing list */
List cons(String s, List list)
{
  List node = (List) malloc(sizeof(NodeStruct));

  node->value = (String) malloc (strlen(s) + 1);
  strcpy(node->value, s);
  node->next = list;
  return node;
}
```



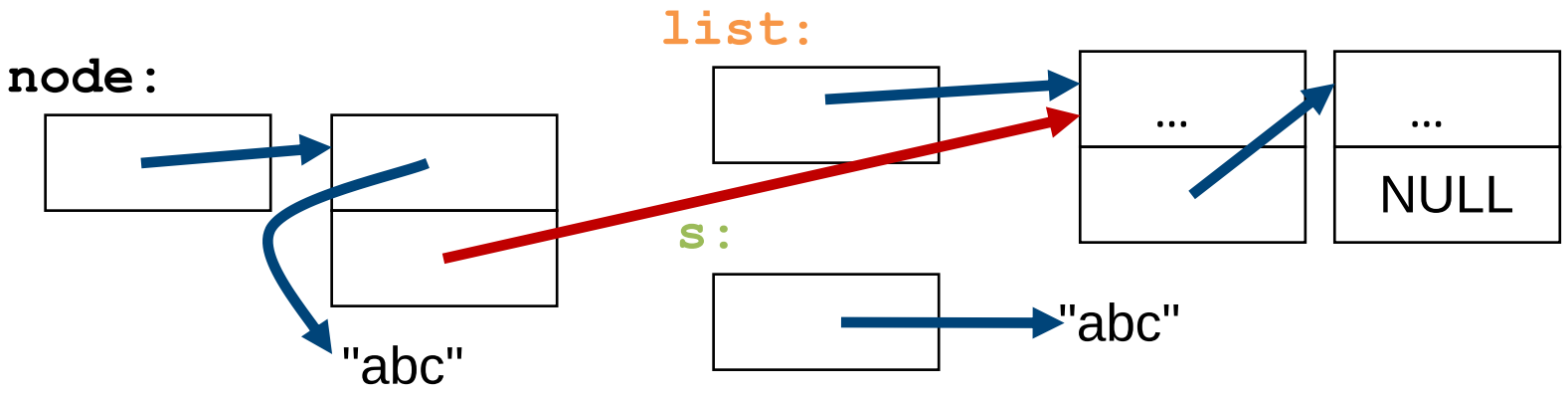
Linked List Example

```
/* Add a string to an existing list */
List cons(String s, List list)
{
  List node = (List) malloc(sizeof(NodeStruct));

  node->value = (String) malloc (strlen(s) + 1);
  strcpy(node->value, s);
  node->next = list;
  return node;
}
```



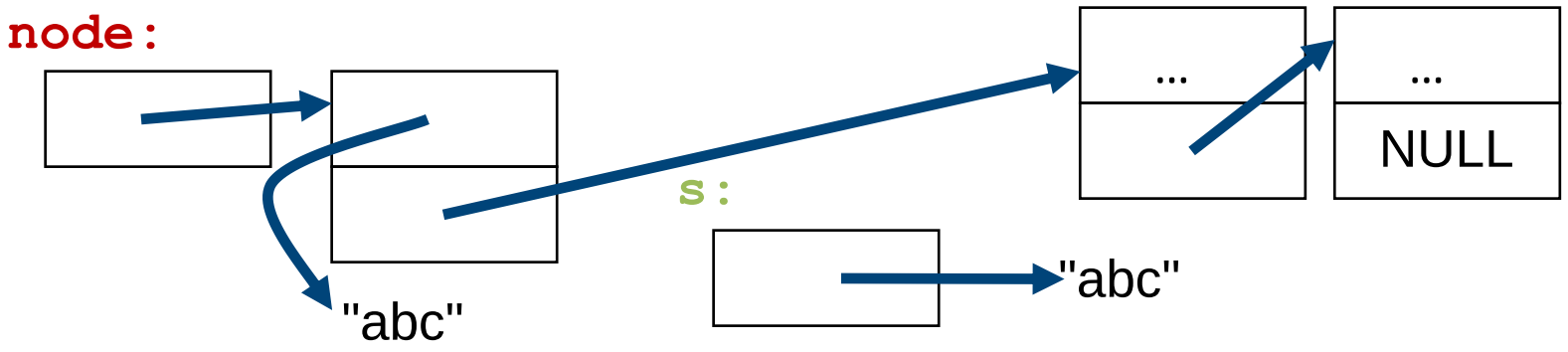
Linked List Example

```
/* Add a string to an existing list */
List cons(String s, List list)
{
  List node = (List) malloc(sizeof(NodeStruct));

  node->value = (String) malloc (strlen(s) + 1);
  strcpy(node->value, s);
  node->next = list;
  return node;
}
```



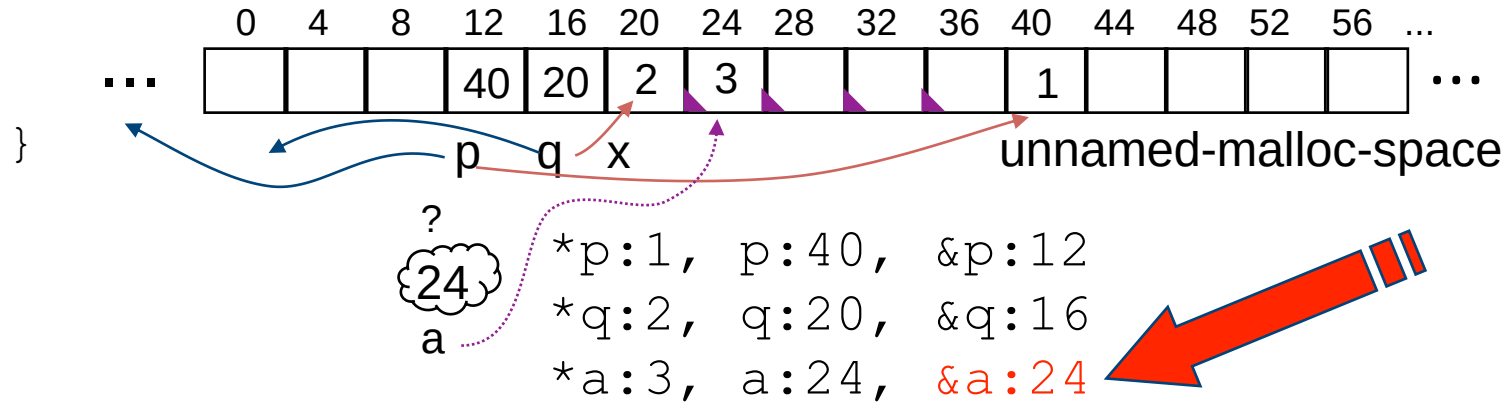
Linked List Example

```
/* Add a string to an existing list */
List cons(String s, List list)
{
  List node = (List) malloc(sizeof(NodeStruct));

  node->value = (String) malloc (strlen(s) + 1);
  strcpy(node->value, s);
  node->next = list;
  return node;
}
```



Revisiting Arrays

```
void foo() {
  int *p, *q, x;
  int a[4];
  p = (int *) malloc (sizeof(int));
  q = &x;

  *p = 1; // p[0] would also work here
  printf("*p:%u, p:%u, &p:%u\n", *p, p, &p);
  *q = 2; // q[0] would also work here
  printf("*q:%u, q:%u, &q:%u\n", *q, q, &q);
  *a = 3; // a[0] would also work here
  printf("*a:%u, a:%u, &a:%u\n", *a, a, &a);
```



K&R: “An array name is not a variable”

Don't forget the globals!

- Remember:
  - Structure declaration does not allocate memory
    - » Only when you instantiate it.
  - Variable declaration does allocate memory
- So far, we have talked about several different ways to allocate memory for data:
  1. Declaration of a local variable in a function (statically)

```
int i; struct Node list; char *string;
int ar[n];
```

  2. "Dynamic" allocation at runtime by calling allocation function (malloc).

```
ptr = (struct Node *) malloc(sizeof(struct Node)*n);
```

- One more possibility exists…
  3. Data declared outside of any procedure/function (i.e., before `main`).
  - Similar to #1 above but has "global" scope.

```
int myGlobal;
main() {
        …
}
```

Useful in C, but not in Java/C++

C Memory Management

- C has 3 pools of memory (based on the nature of usage)
  - Static storage: global variable storage, basically permanent, entire program run
  - The Stack: local variable storage, parameters, return address (location of "activation records" in Java or "stack frame" in C)
  - The Heap (dynamic malloc storage): data lives until deallocated by programmer

- C requires knowing where things are in memory, otherwise things don't work as expected
  - Java hides location of objects

Normal C Memory Management

- A program's address space contains 4 regions:
  - stack: local variables, grows downward
  - heap: space requested for pointers via `malloc()` ; resizes dynamically, grows upward
  - static data: Initialized/uninitialized static and global variables
  - code/text: loaded when program starts, does not change

For now, OS somehow prevents accesses between stack and heap (gray hash lines). Wait for virtual memory