COMP 206 – Introduction to Software Systems

Lecture 16 – C struct and Linked Lists

Date: October 31st, 2018

Outline

- Introduce the C struct – a handy way to organize many variables!
- typedef and the common struct syntax
- Allocating memory for structs
- Introducing complicated C data types

Organizing Many Types of Data

- So far arrays let us hold the same single type of value in each entry

- What about the many cases where we want to group related elements that have a variety of data types and meanings?
  - Bank account with balance, account number, owner's name, bank address, list of transactions
  - Amazon order with buyer info, seller info, product details, shipping details
  - Class list with student ID, first and last names, list of grades, code submissions

In Other Languages

- Python:
  - We could consider using a dictionary. Keys indicate the name of the data and values indicate the contents. E.g., my_account = { “type” : “checking”, “number” : 1234567, “balance” : “one MILLION dollars” }
  - Even more useful a class with all of the same data attributes as well as functions such as “def transfer(self, other, amount)”
- Java:
  - A class is likewise our most likely solution
- C++:
  - Has all of the C functionality plus several additional features, but mainly classes
- C does not provide object oriented functionality (no class keyword), but we can achieve must of this by ourselves if we’re creative. That’s next!

The C Struct

```
struct TYPE_NAME    // optional user defined identifier
{
FIELD1;          // TYPE VAR;   not initialized
FIELD2;
:
FIELDn;

} VAR_NAME;         // optional identifier variable



// Example declarations
struct TYPE_NAME var1, var2, array[10], *p;

// Example usage
var1.FIELD1 = ...
array[3].FIELD2 = ...
printf( "...", p->FIELD3 );
```

The 206 Course Example

- Define a structure that holds info about our course:

```
struct COURSE {
  unsigned int numberOfStudent;
  char nameProfessor[100];
  char buildingName[100];
  unsigned int roomNumber;
}
```

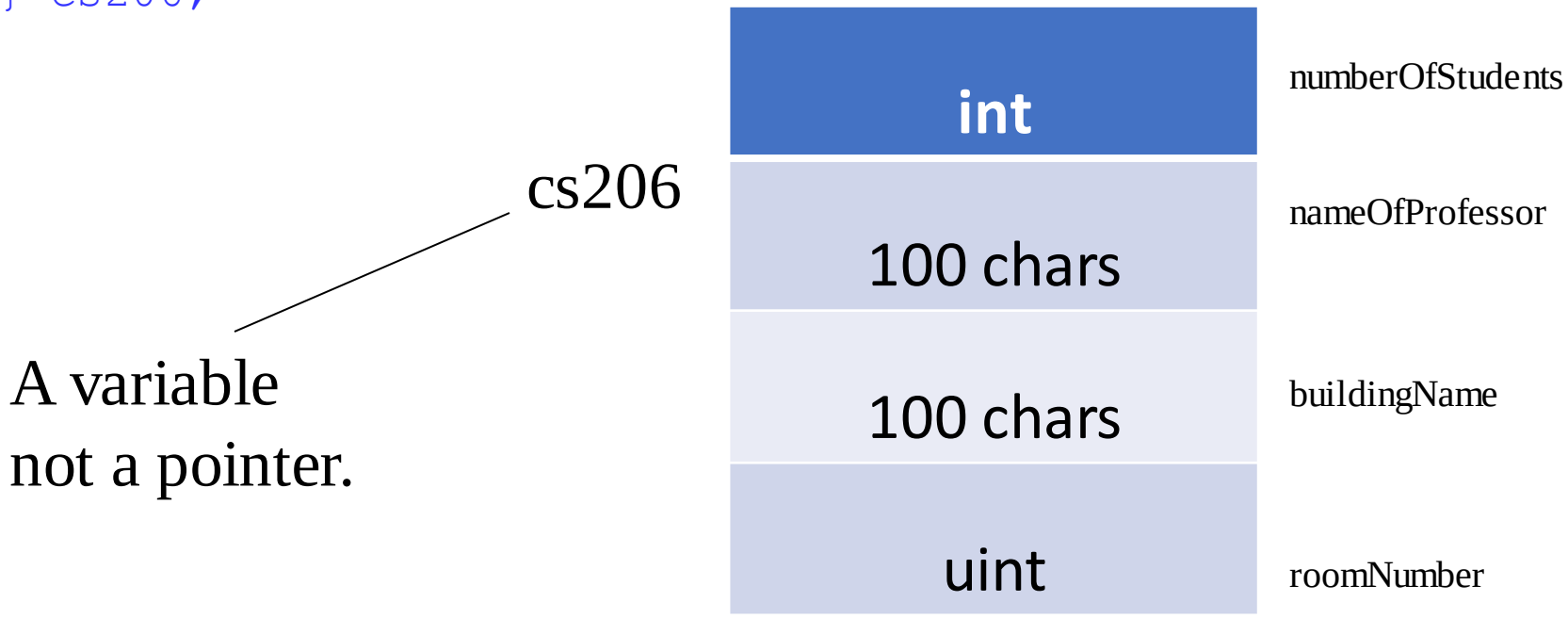
- Now we're able to use a new type name "struct COURSE".
- We need both struct and COURSE keywords for now as this is the C syntax to help us remember this is a type we created.

Using stucts

- Create an empty variable:
  - struct COURSE cs206;
- And fill it with data this way:
  - cs206.numberOfStudent = 530;
  - strcpy(cs206.nameOfProfessor, “A spooky ghost”);
  - strcpy(cs206.buildingName, “Maass”);
  - cs206.roomNumber = 10;
- Or do both at once:
  - struct COURSE cs206 = { 530, “A spooky ghost”, “Maass”, 10 };

Creating a variable immediately

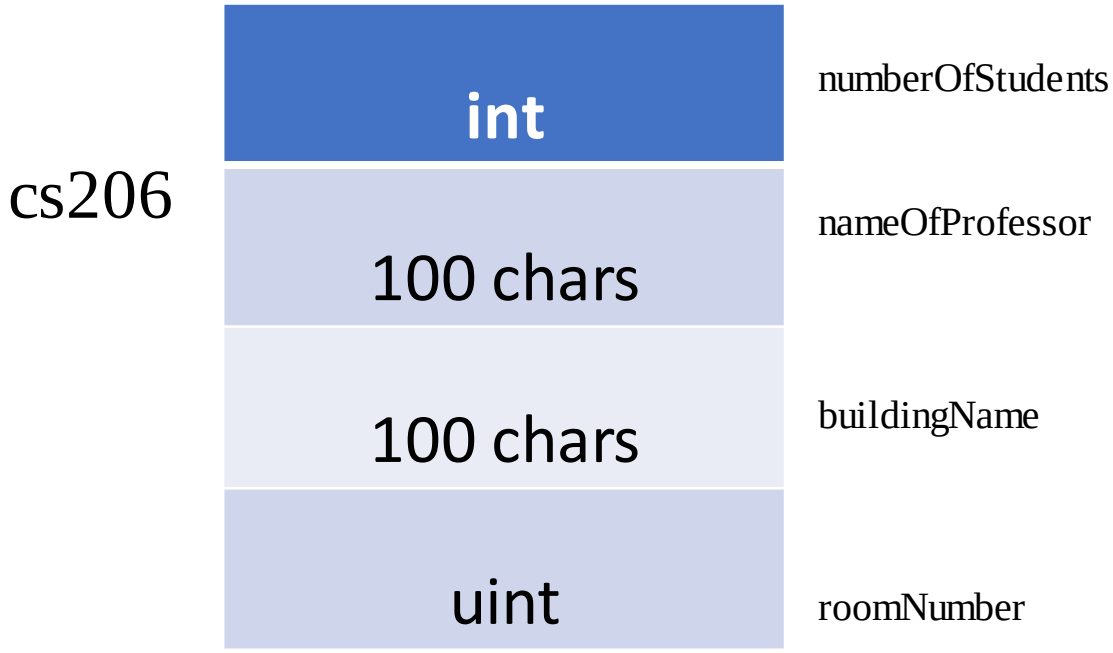
```
struct COURSE{
    int numberOfStudents;
    char nameOfProfessor[100];
    char buildingName[100];
    unsigned int roomNumber;

} cs206;
```



Creating a variable immediately

```
struct COURSE{
    int numberOfStudents;
    char nameOfProfessor[100];
    char buildingName[100];
    unsigned int roomNumber;

} cs206;
```

Including a name here creates a variable (takes memory)

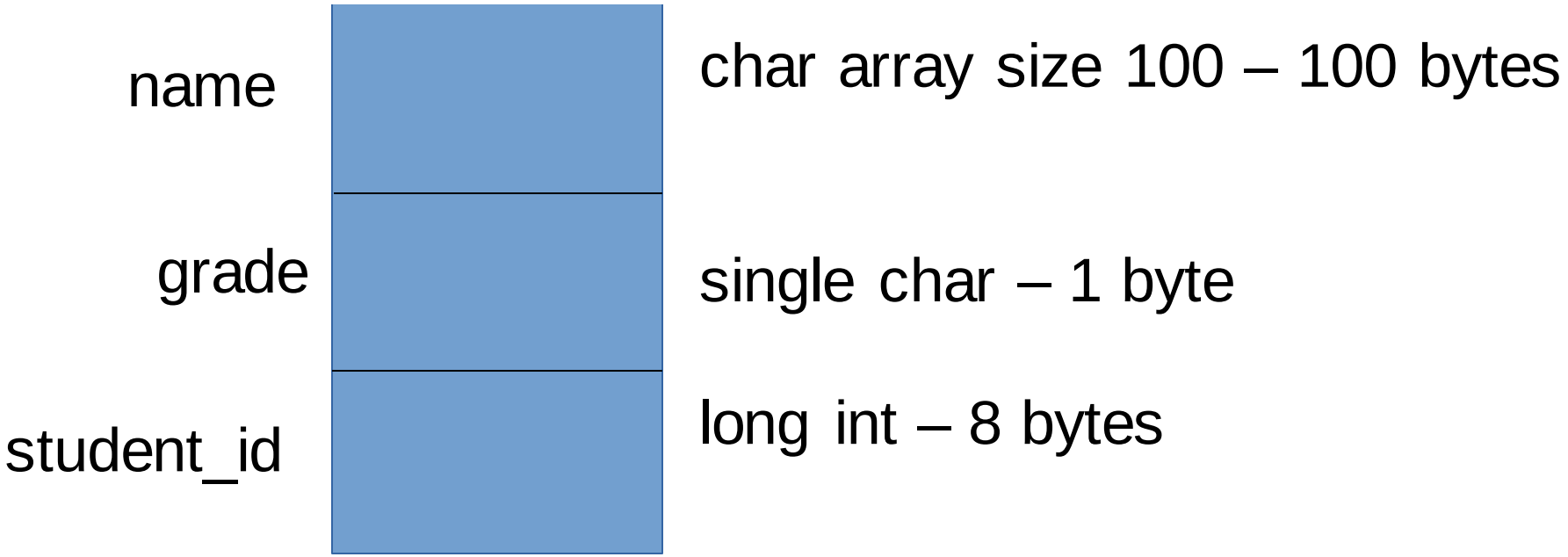


How does the struct live in memory?

```
struct student{

  char name[100];
  char grade;
  long int student_id;
};

struct student dave = { "Dave", '?', 2600000000 };
```

So, what do we expect from this line?

```
printf( "The size of the struct is %ld.\n", sizeof(struct student) );
```



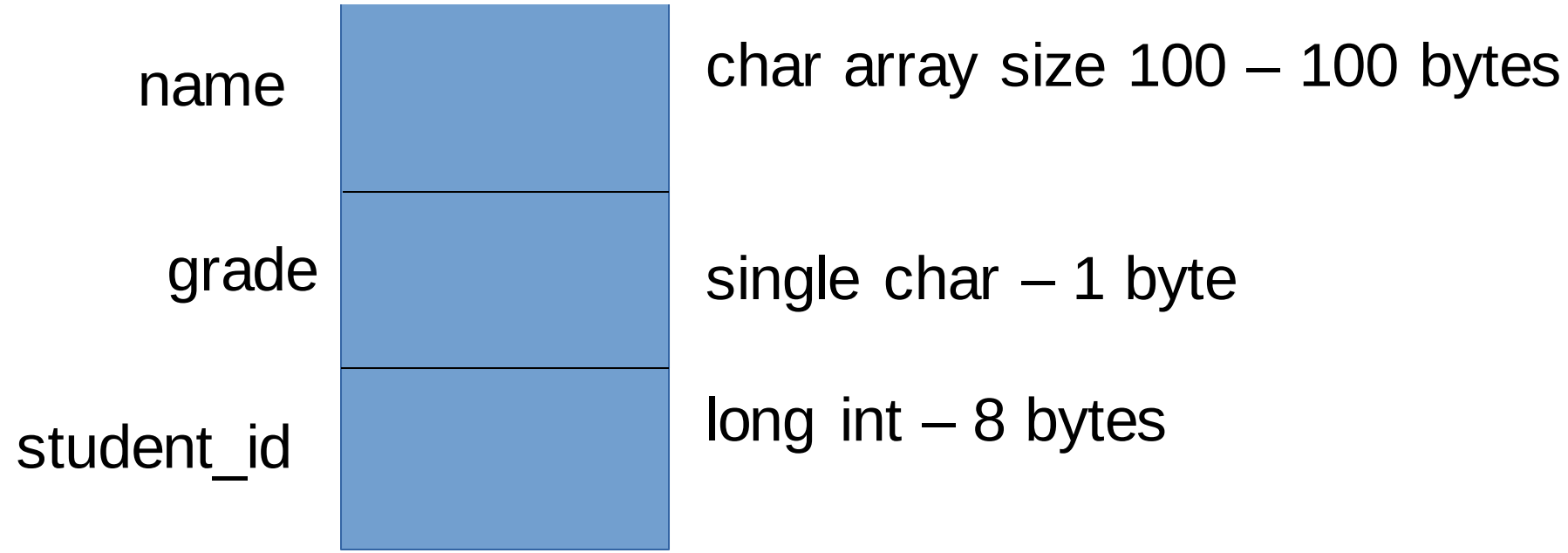
How does the struct live in memory?

```c
struct student{

   char name[100];
   char grade;
   long int student_id;
};

struct student dave = { "Dave", '?', 2600000000 };
```

So, what do we expect from this line?

```c
printf( "The size of the struct is %ld.\n", sizeof(struct student) );
```

"The size of the struct is 112" - Why?



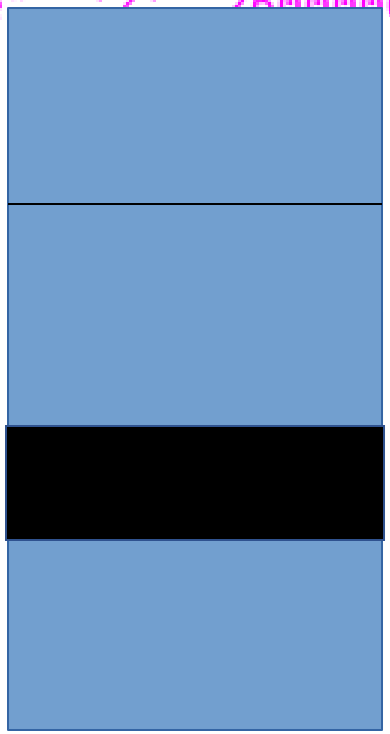
How does the struct live in memory?

```
struct student{

   char name[100];
   char grade;
   long int student_id;
};

struct student dave = { "Dave"  '2'  2600000000 };
```

So, what do we expect from this line?

```
printf( "The size of the struct is %ld.\n", sizeof(struct student) );
```

"The size of the struct is 112" - Why?



name — char array size 100 – 100 bytes

grade — single char – 1 byte

padding — 3 bytes to pad at multiple of 4!

student_id — long int – 8 bytes

What is dangerous with array memory?

```
struct student{

  char name[100];
  char grade;
  long int student_id;
};

struct student dave = { "Dave", '?', 2600000000 };
```

- Any code that assumes no padding and hard-codes.
  - E.g., long int *id_ptr = (long int*)((*char)(&(dave)) + 101);
- Also, any code that assumes the padding and hard-codes.
  - E.g., long int *id_ptr = (long int*)((*char)(&(dave) + 104);
- In fact, it is best not to assume the struct ordering!

What is better?

- Use the struct names and the sizeof command itself
  - sizeof(struct student) to get the overall size, never hardcoded
  - &(student.student_id) to get the address that starts a particular field

- This way, the compiler protects you and your code runs no matter what changes in the struct (suppose you decide to add a field and re-compile!)

Structs can live inside or outside of main()

```
int main(){
    struct ABC{
        int x; int y;
    } a;

     a.x = 10;
}
```

```
struct ABC
{
    int x, y;
} a;

int main() {
    a.x = 10;
}
```

typedef and struct

- Always typing "struct COURSE" for the type is a bit annoying
- You can use typedef to define a new type name:

```
typedef struct COURSE {
    int number_of_students;
    char name_professor[100];
    char location_building[100];
    int location_room;
} MYCOURSE;
```

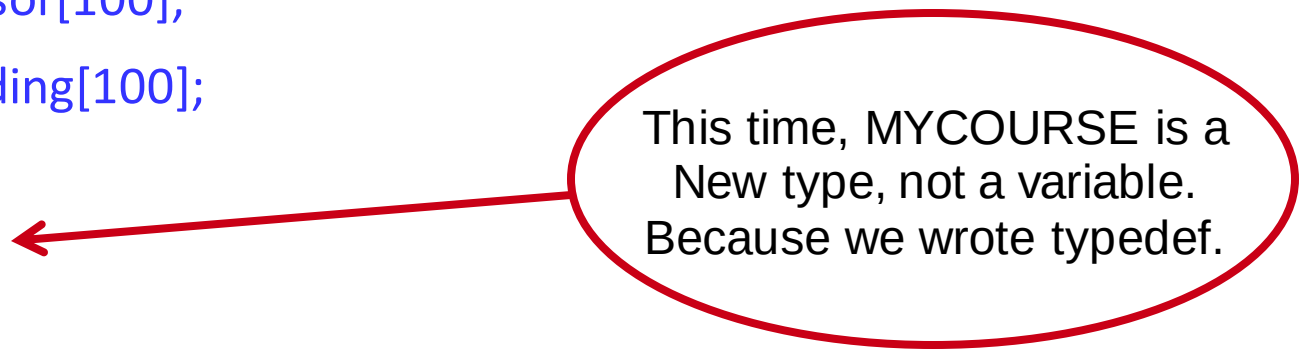
- When creating a new variable of this type, you no longer need to specify the struct keyword:

```
MYCOURSE cs206;
```

typedef and struct

- Always typing "struct COURSE" for the type is a bit annoying
- You can use typedef to define a new type name:

```
typedef struct COURSE {
    int number_of_students;
    char name_professor[100];
    char location_building[100];
    int location_room;
} MYCOURSE;
```



- When creating a new variable of this type, you no longer need to specify the struct keyword:

```
MYCOURSE cs206;
```

Question

1. How do we build an array of structures?
2. How do we use it to store and change some data?

Struct array syntax example #1

```
struct student_rec{
    char name[100];
    int age;
    float gpa;
} students[50];

students[2].age = 20;

gets(student[2].name);

for(i=0;i<50;i++){
    printf("%s",students[i].name);
}
```

Struct array syntax example #2

```
struct COURSE {
    int number_of_students;
    char name_professor[100];
    char location_building[100];
    int location_room;
};
struct COURSE mcgill[10];
mcgill[5].noofstudents = 220;
strcpy(mcgill[5].nameofprof, “bob”);
```

Memory view on array of structs

```
struct PERSON
{
        char name[100];
        int age;
};

struct PERSON x[10]; // array

x[1].age = 19;
x[9].name = “Bob”;
printf(“%s”, x[9].name);

x[9].name[2] = 'A';
```

Indexing into array with [i] gets a single struct. Indexing with "." gets a single field.

What about pointers? A new syntax

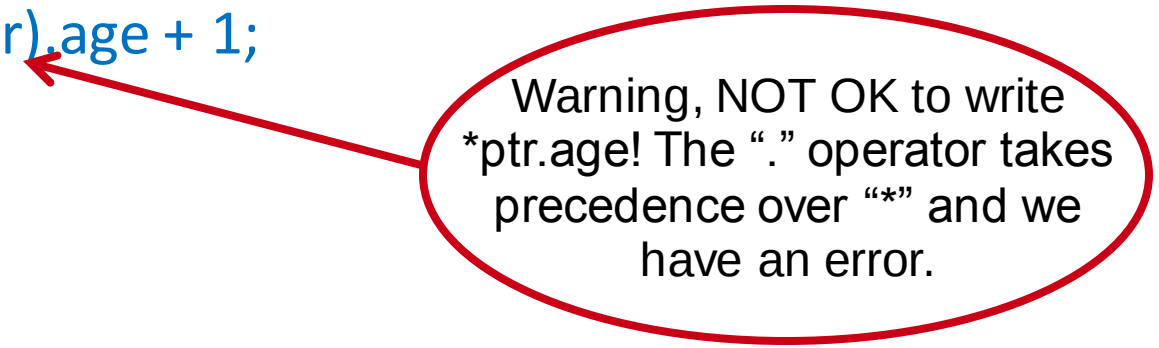
- The "arrow" operator is something new in C. Just designed for pointers to structs, it combines "dereference" (*) and "access struct field" (.)

```
struct PERSON me = { “David”, -1 };
struct PERSON *ptr = &me;
ptr->age = ptr->age + 1; // Equivalent to:
(*ptr).age = (*ptr).age + 1;
```

What about pointers? A new syntax

- The "arrow" operator is something new in C. Just designed for pointers to structs, it combines "dereference" (*) and "access struct field" (.)

```
struct PERSON me = { “David”, -1 };
struct PERSON *ptr = &me;
ptr->age = ptr->age + 1; // Equivalent to:
(*ptr).age = (*ptr).age + 1;
```



How about malloc of a struct?

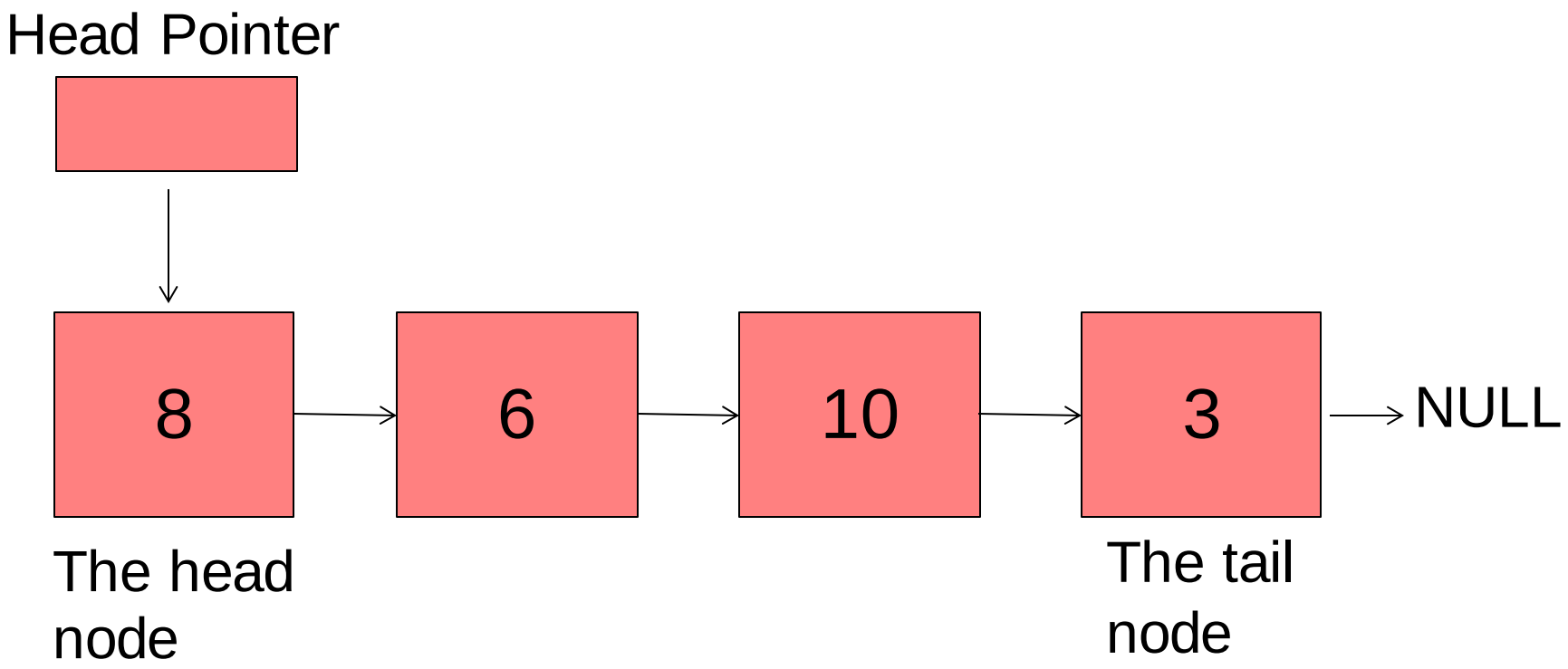
```
struct STUDENT
{
    char name[50];
    int age;
    double gpa;
};

struct STUDENT *p;

p = (struct STUDENT *) malloc(sizeof(struct STUDENT));

p->age = 18;

p->gpa = 3.5;

strcpy(p->name, "Mary Smith");

free(p);
p = NULL;
```

So, how is this new tool really used?

- Lots of the examples we gave earlier on, just to group data:
  - Courses, people, bank accounts
- Maybe more important: core CS data structures
  - Tree, list, queue, graph, etc.

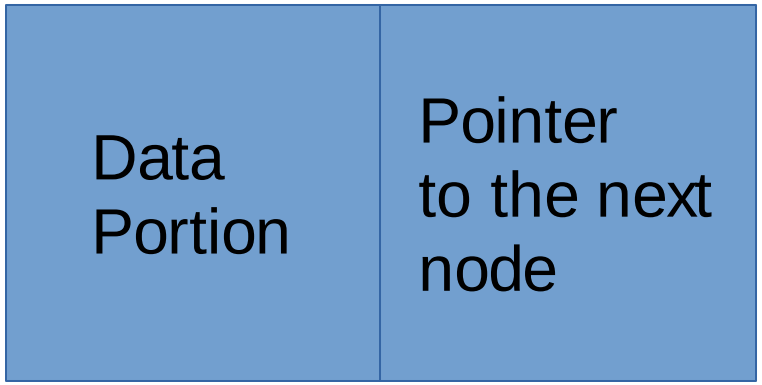
Example: The Linked List

A link list starts with a head pointer and ends with a NULL.



Each data box is called a NODE:
holds a value and a "next node" pointer

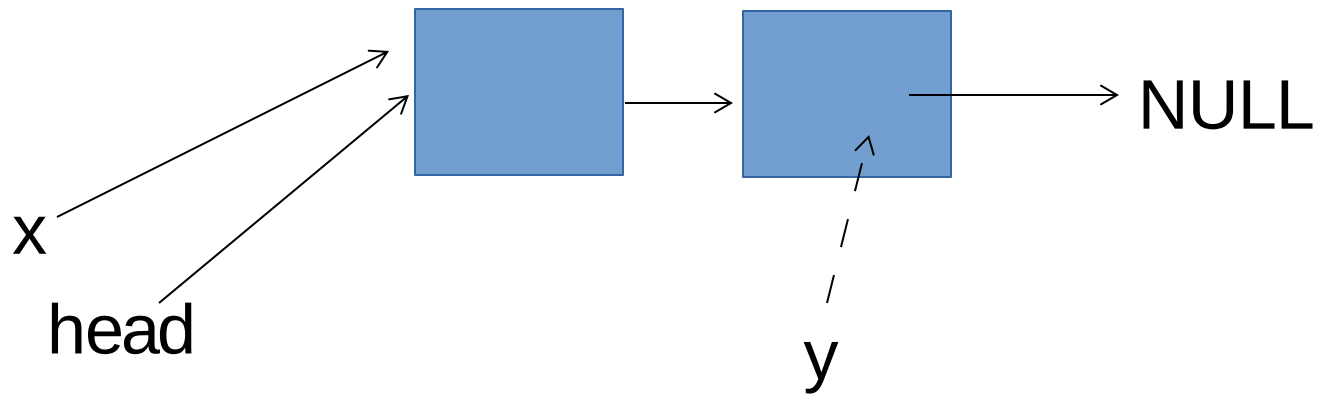
A NODE is made from 2 parts



A Node

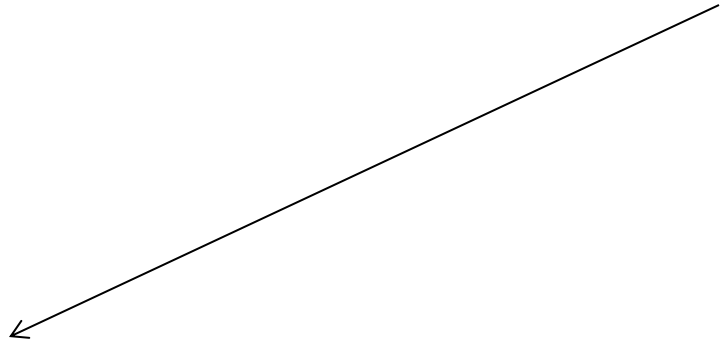
The last node has NULL in its pointer.

```
typedef  struct PERSON
{
        char name[100];
        int age;
        struct PERSON *next;
} s_person;

s_person *x = (s_person*)malloc(sizeof(s_person));
s_person *y = (s_person*)malloc(sizeof(s_person));
s_person *head;

x->next = y;
y->next = NULL;
y=NULL;
head = x;
```



Generic Linked List Pattern

```
/* Node for the link list */

typedef struct NODE
{
        int value;
        struct NODE* next;

} aNode;
```



Int is only one example. This could also be many fields, or even be another struct.

Example C main

```
typedef struct NODE
{
        int value;
        struct NODE* next;

} aNode;

aNode *head=NULL;

int main(void) {
   printLinkedList(head);
   head = addToLinkedList(head, 5);
   head = addToLinkedList(head, 10);
   printLinkedList(head);
}
```

```
Desired output:
$ ./a.out
List is empty.
Content of list is : 5 10
```

Test: Can we add to our list like this?

```
void addToLinkedList(aNode* list, int value) {

        aNode* freeSpot;
        aNode* newNode;

        // Find a free spot at the end to add the value
        freeSpot = list;
        while(freeSpot->next != NULL) {
                freeSpot = freeSpot->next;
        }

        newNode = (aNode *)malloc(sizeof(aNode));
        newNode->value = value;
        newNode->next  = NULL;
        freeSpot->next = newNode;
}
```

Test: Can we add to our list like this?

```
void addToLinkedList(aNode* list, int value) {

        aNode* freeSpot;
        aNode* newNode;

        // Find a free spot at the end to add the value
        freeSpot = list;
        while(freeSpot->next != NULL) {
                freeSpot = freeSpot->next;
        }

        newNode = (aNode *)malloc(sizeof(aNode));
        newNode->value = value;
        newNode->next  = NULL;
        freeSpot->next = newNode;
}
```

Problem:

1) list can be NULL and this is not checked before using

2) Even if we fix this, changing the value of "freeSpot" does not change the pointer value in main (PASS BY VALUE)

```
aNode* addToLinkedList(aNode* list, int value) {
   aNode* freeSpot;
   aNode* newNode;
   freeSpot = list;

   if( list == NULL ){ // First item added
      newNode = (aNode*)malloc(sizeof(aNode));
      newNode->value = value;
      newNode->next = NULL;
      return newNode;
   }
   // Find a free spot at the end to add the value
   while(freeSpot->next != NULL) {
       freeSpot = freeSpot->next;
   }
   newNode = (aNode *)malloc(sizeof(aNode));
   newNode->value = value;
   newNode->next  = NULL;
   freeSpot->next = newNode;

   return list;
}
```

```
void addToLinkedList(aNode** list, int value) {


    aNode *newNode = (aNode *)malloc(sizeof(aNode));
    newNode->value = value;
    newNode->next  = NULL;

    aNode* freeSpot = *list;

    if( freeSpot == NULL ) *list = newNode;
    else {
        while(freeSpot->next != NULL) {
            freeSpot = freeSpot->next;
        }
        freeSpot->next = newNode;
    }
}
```

Pretty Print the linked list

```
/* Pretty print the list. */

void printLinkedList(aNode* list) {

        if (list != NULL) {
                printf("Content of list is :");
                printNodes(list);
        } else {
                printf("List is empty.");
        }

        printf("\n");

}
```

Printing the nodes

```
void printNodes(aNode* my_node) {

        printf(" %i ", my_node->value);
        if (my_node->next != NULL) {
                printNodes(my_node->next);
        }
}
```

Exercises

- Insert a single new node into a sorted list:
- insert_sorted(node** list, int new_value);

- Add one list to the end of another with:
- list_append(node** dest, node* source);

- Delete all nodes that have matching value:
- delete_by_value(node** list, int value);