EEE101: C Programming & Software Engineering I

Lecture 8: Memory & Structures

Dr. Rui Lin/Dr. Mark Leach
Office: EE512/EE510
Email: rui.lin/mark.leach@xjtlu.edu.cn
Dept. of EEE XJTLU

Outline of Today's Lecture (8)

- Dynamically Allocating/Freeing Memory
- Introduction to Structures
- Using Structures in a Data Structure
- Linked Lists
- Structures and functions

Memory Allocation

- The compiler allocates enough memory for your whole program including variables, arrays and functions.
- But what happens if you need more memory space during program execution?

  C allows for memory to be allocated at runtime

  i.e. you can add memory space on demand (as long as it is available)

The malloc() function (1/3)

- malloc() requests a **block of memory** to be made available of a specified size
- The return value of the function is a pointer to the newly available memory
- The type of the pointer is generic

  Needs to be typecast to the required type (e.g. int, float, char etc..)

  Note malloc() is contained in stdlib.h

The malloc() function (2/3)

Note the typecast
(**double** * ptr)

Input parameter
size of memory needed
6 **double**'s

```
ptr = (double*) malloc(6*sizeof(double));
```

- malloc() finds a suitable block of free memory and returns the address of the first byte
- That address is assigned to a pointer variable (ptr)
- typecast is needed to make sure that the ptr type and malloc() return type are the same

The malloc() function (3/3)

- If the function fails to obtain the memory NULL (‘\0’) is returned
- Usually the malloc() function is used with a test for failure.

```
if (ptr == NULL)
        {
        printf (“Memory allocation Failed”);
        return 1;
        }
```

The free() function (1/2)

- To release memory allocated with malloc(), use:

```
free(ptr);
```

- Frees the block of memory starting at the address stored in ptr.
- Should ONLY be used where the memory block was allocated by malloc()
- **You cannot use free() to free memory allocated by other means**

Note free() is contained in stdlib.h

Creating dynamically sized arrays

Two ways to create an array

- Declare an array using constant expressions for the array dimension i.e. **int** array[10]. Once declared the size of this array is fixed. It cannot be resized at runtime.
- Declare a pointer then call malloc() with the number of elements you want determined at runtime. The pointer can be used to access the elements of the array. The length of the array is then flexible each time you run the program. The array is said to be dynamically sized.

Example program (1/2)

```
#include <stdio.h>
#include<stdlib.h> /*malloc() and free()*/
int main(){
double *ptd;
int max, number, i=0;
puts("What is the number of double entries?");
scanf("%d",&max);
ptd = (double *)malloc(max * sizeof(double));
if(ptd==NULL){
        puts("Memory allocation failed. Goodbye.");
        return 0;
        }
/*ptd now points to an array of doubles of length max*/
```

Example program (2/2)

```
puts("Enter values or q to quit");
while (i < max && scanf("%lf",&ptd[i])==1)
        i++;
printf("Here are your %d entries: \n", number = i);
for(i=0; i<number; i++){
        if(i%10 == 0)
                putchar('\n'); /*print 10 numbers per line*/
        printf("%lf", ptd[i]);
        }
puts("\nDone");
free(ptd);                  /*free the memory*/
return 0;
}
```

Structures…What and Why? (1/2)

- Most objects in the world are identified by a collection of data, in which each member data defines one aspect of the object.
- Each member data may be a different type e.g. You as a person may be defined by:
  - Name (**char**aracter string)
  - ID number (**int**eger)
  - Age (**int**eger)
  - Profession (**char**acter string) etc…etc.

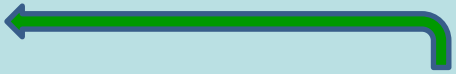
Structures…What and Why? (2/2)

- C provides a way of defining a giant data type that allows many member data to be combined. The whole data set is its own data type or object.
- This data type is called a structure
- Inside a structure a set of members is declared

Creating a Structure

- Structures are declared using the keyword **struct**
- The keyword can be followed by an optional name (e.g. book), to identify the form of the structure.
- The content of the structure is then defined in {};

```
#include<stdio.h>
struct book{
        char title[20];
        char author[20];
        float value;
        };
      /*Don't miss the ;*/
```



Note: This is like declaring a new data type called book. It has not created any variables of this type…

Declaring a Structure

- Now we have a structure we need to declare a variable of this type:

```
#include<stdio.h>
struct book{
        char title[20];
        char author[20];
        float value;
        };
main(){
struct book my_book;
/*my_book is now a
structure*/
```

```
#include<stdio.h>
struct book{
        char title[20];
        char author[20];
        float value;
        }my_book;
main(){
…
/*alternative declaration*/
```

Declaring a Structure with typedef

- For convenience the keyword **typedef** can be used.

```
#include<stdio.h>
typedef struct book{
        char title[20];
        char author[20];
        float value;
        }lib_book;

main(){
lib_book name;
```

Note: lib_book becomes a variable type and can be used like int instead of writing struct book

Initialising a Structure

- Initialisation can be by a ‘,’ separated list of values.

```
#include<stdio.h>
struct book{
        char title[20];
        char author[20];
        float value;
        };
main(){
struct book my_book={
        “Dracula”,
        “Bram Stoker”,
        9.99};
```

Note: Do not assume the memory size of the structure is the sum of the sizes of its members. It is usually larger.
Use “sizeof(book)” to find actual size.

Accessing a Structure

- Use the dot (.) operator to refer to a structure member
- members can then be used like any other variable

```
struct book{
        char title[20];
        char author[20];
        float value;
        };
main(){
struct book my_book={
        "Dracula",
        "Bram Stoker",
        9.99};
float price;

price = my_book.value;

printf("the book price is
                %d",price);
```

Filling/Accessing a Structure (1/5)

- The content of one book is input from the keyboard

```
#include<stdio.h>
struct book{
        char title[20];
        char author[20];
        float value;};
main(){
struct book lib;
gets(lib.title);
gets(lib.author);
scanf("%f",&lib.value);
printf("%s by %s is £%.2\n", lib.title, lib.author, lib.value);}
```

Declaration not creation. book ONLY describes structure content

Don't forget the ;

Creation of object

Filling/Accessing a Structure (2/5)

- The content of one book is input from the keyboard

```
#include<stdio.h>
struct book{
        char title[20];
        char author[20];
        float value;};
main(){
struct book lib;
gets(lib.title);
gets(lib.author);
scanf("%f",&lib.value);
printf("%s by %s is £%.2\n", lib.title, lib.author, lib.value);}
```

Keyword struct identifies that the following is a structure variable

book is the name of the structure

Content can be a mix of as many variables and types as wanted (even another structure).

Filling/Accessing a Structure (3/5)

- The content of one book is input from the keyboard

```
#include<stdio.h>
struct book{
        char title[20];
        char author[20];
        float value;};
main(){
struct book lib;
gets(lib.title);
gets(lib.author);
scanf("%f",&lib.value);
printf("%s by %s is £%.2\n", lib.title, lib.author, lib.value);}
```

Create a structure of type book called lib

To gain access, use the name of the struct variable followed by the dot then the name of the member i.e. name.member

Filling/Accessing a Structure (4/5)

- The content of one book is input from the keyboard

```
#include<stdio.h>
struct book{
        char title[20];
        char author[20];
        float value;};
main(){
struct book lib;
gets(lib.title);
gets(lib.author);
scanf("%f",&lib.value);
printf("%s by %s is £%.2\n", lib.title, lib.author, lib.value);}
```

gets() copies a string from the keyboard and places it at an address. The member title is an array, remember the name of an array is the address of the first element

Filling/Accessing a Structure (4/5)

- The content of one book is input from the keyboard

```
#include<stdio.h>
struct book{
        char title[20];
        char author[20];
        float value;};
main(){
struct book lib;
gets(lib.title);
gets(lib.author);
scanf("%f",&lib.value);
printf("%s by %s is £%.2\n", lib.title, lib.author, lib.value);}
```

scanf() is used to get a real number from the keyboard. '&' is required since value is a float variable

member values are accessed with the dot (.) operator

Pointers to Structures

- Yes….they are back again!
- Pointers allow a locally defined **struct** to be modified in a function.

```
main(){
char *title;
struct book my_book;
struct book *pbook;
pbook = &my_book;
title = pbook ->title;}
```

declaration uses the usual * type is struct name
pbook is pointer name

& gives address of the struct variable my_book

Access to members using pointers requires
'->' or (*pbook).title not '.'

Using Pointers to Structures

```
#include<stdio.h>
struct book{
        char title[20];
        char author[20];
        float value;};
main(){
struct book lib;
struct book *plib=&lib;
gets(plib->title);
gets(plib->author);
scanf("%f",&plib->value);
printf("%s by %s is £%.2\n", plib->title, plib->author,
                                        plib->value);}
```

plib is a pointer to the address of lib

Note . and -> have the highest precedence of ALL operators

What about more than 1 book?

Arrays of Structures

```
#include<stdio.h>
struct book{
        char title[20];
        char author[20];
        float value;};

main(){
struct book lib[10];
gets(lib[1].title);
gets(lib[1].author);
scanf("%f",&lib[1].value);
printf("%s by %s is £%.2\n", lib[1].title, lib[1].author,
                                        lib[1].value);}
```

Declaration statement declares an array

Array starts at lib [0]. Array elements accessed as usual i.e. second structure lib[1]

Beyond Arrays, Linked Lists (1/3)

- Normally your program design will require many decisions before coding begins. The **linked list** is a fundamental form of data structure.
- Suppose you need a program to form a list of your favourite movies. A movie has a variety of info. e.g. title, director, year, genre etc. This suggests a **struct**.
- But how many would you like?
- **Could use an array**
  - **Enables direct use of pointers to move through the array**

What problems do arrays present?

Beyond Arrays, Linked Lists (2/3)

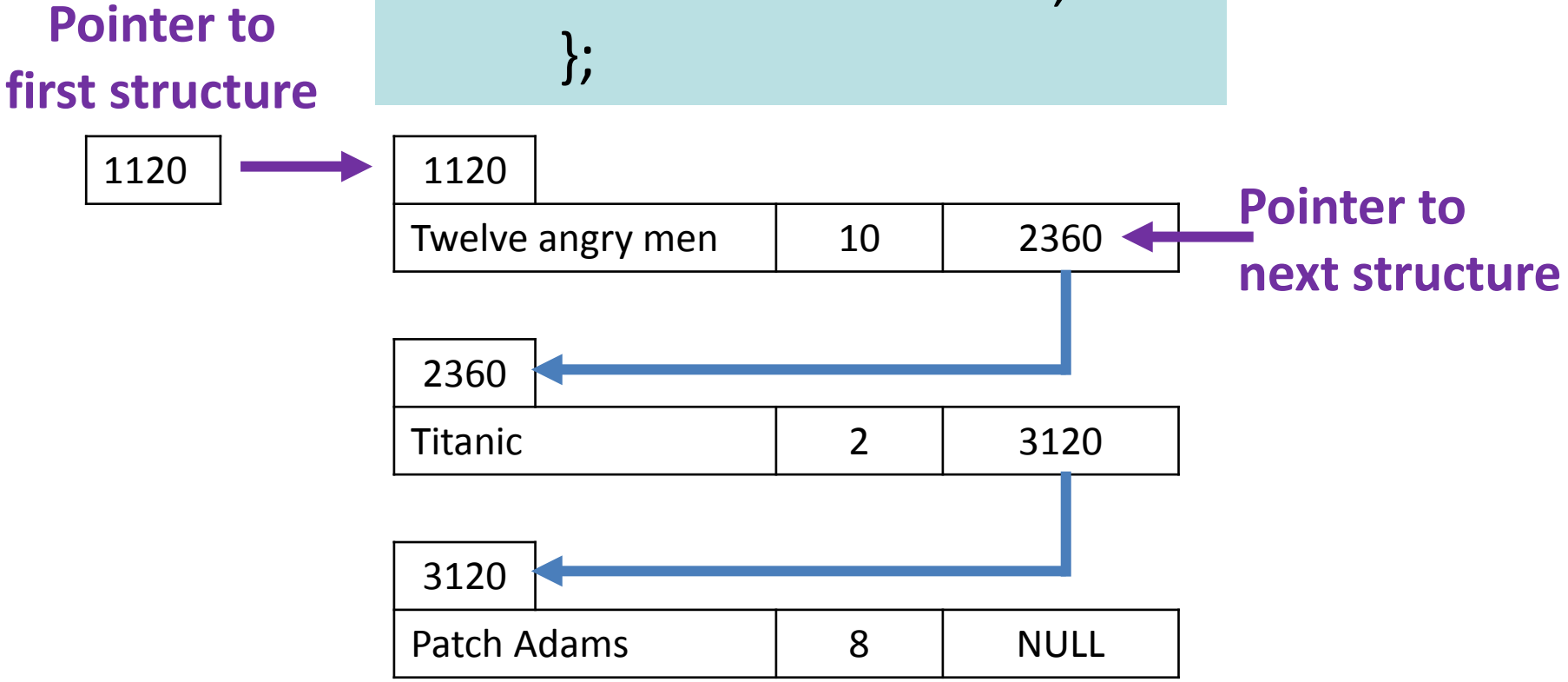
- Problems with an array of structures :
  - **The array is fixed in size, not expandable**
  - **Difficult to sort the array in a given order**
  - **Difficult to delete an entry (maybe you change your mind)**
- Any other choice?
- Dynamically create new structures when required using malloc(), allows indefinite expansion
- Any difficulties?
- **Need to link the structures together**

Linked Lists (1/3)

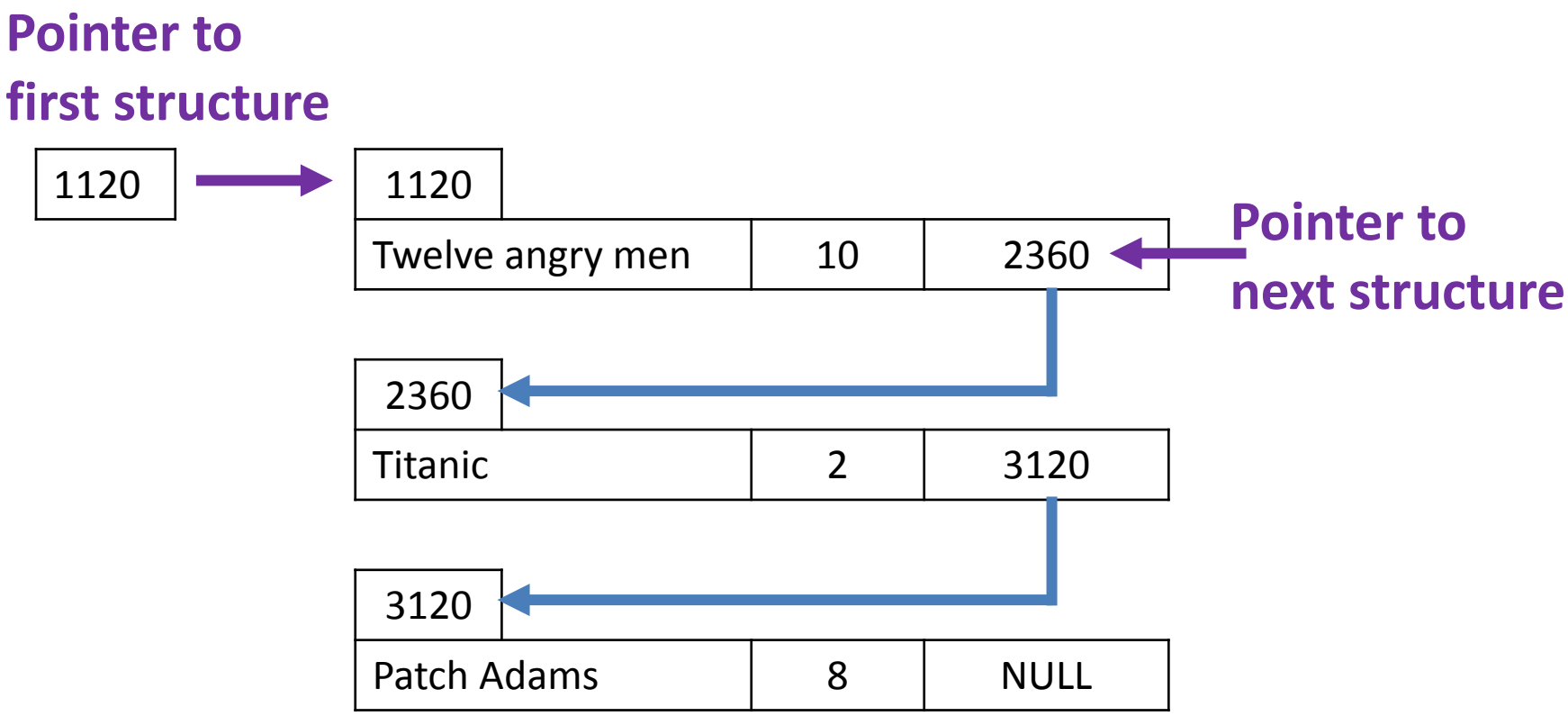
- Each time malloc() is called, a pointer to the new memory elements is created.
- These will not usually be created in any particular location order therefore a simple pointer cannot be used to move from one structure to the next
- The linked list includes a pointer variable inside the structure, which is set to point to the next structure in the list.

Linked Lists (2/3)

```
struct movie{
        char * title;
        int rating;
        struct movie *next;
        };
```



Linked Lists (3/3)



How could an element be inserted into or deleted from, the list?

Structures as Arguments (1/3)

- Structures can be used as arguments

```
myprint(mystructure);
```

  - In this case a copy of the structure “mystructure” is passed to the function “myprint”
  - This can be slow if the structure has many members or contains large arrays
- This is different from passing an array to a function, which passes a pointer.

Structures as Arguments (2/3)

```
#include<stdio.h>
struct funds{          /*declare the structure*/
        double current;
        double savings;
        };
double sum(struct funds);

main(){                /*function call sends copy*/
struct funds ABC;                    /*of ABC*/
ABC.current = 1005;
ABC.savings = 34.5;
printf("Total funds in ABC %lf", sum(ABC));
}
```

Structures as Arguments (3/3)

```
/*function argument is a structure of same type*/
                    ↓
double sum(struct funds my_funds){
        double total;
        total = my_funds.current + my_funds.savings;
        return total;
        }
```

Questions…

- True or False?
  - A **struct** is a built in data type of C
  - A structure is used to declare a data type containing multiple fields (members).
  - Pointers can be used to access members of a structure
  - An array cannot be a structure member

- How is a linked list different from an array?

Questions?

The end of Week 9 already…keep it going we are almost finished ☺