Introduction to C

Session 6

Structures in C

- A ‘struct’ is a collection – but not necessarily of like types
- It defines a new datatype
- It is not an object specification (like classes in C++, C#, Java, etc)
  - A struct does not allow functions - only data members
  - In C, no concept of public/private/data hiding
  - Can get around the function constraint using ‘function pointers’
    - Beyond the scope here though!

Structures in C

A struct can contain other structs as data members

```
struct Point{
    int x;
    int y;
};

int main(int argc, char **argv)
{
    struct Point p;
    p.x = 1;
    p.y = 3;
    printf("%d", p.y);
}
```

Declares a struct datatype called ‘Point’, which contains two ints

Note the closing semi-colon

Defines a variable called ‘p’ of type ‘Point’

Access the data members using ‘.’

Structures in C

```c
struct Point{
    int x;
    int y;
};

int main(int argc, char **argv)
{
    struct Point p;
    p.x = 1;
    p.y = 3;
    printf("%d", p.y);
}
```

This can be declared in a header file, and then defined and used in the same way.

*as long as the header file is included!

Structures in C

```c
typedef struct Pnt{
    int x;
    int y;
} Point;

int main(int argc, char **argv)
{
    Point p;
    p.x = 1;
    p.y = 3;
    printf("%d", p.y);
}
```

We can also use ‘typedef’ to redefine the struct as the alias ‘Point’

This makes it easier and less cluttered to define a struct variable

Pointers and structs

```
Point p;
Point *pp = &p;
pp->x = 1;
pp->y = 3;
//This is the same as above
//p.x = 1;
//p.y = 3;
printf("%d", pp->y);
```

If we create a pointer to the struct

We can access the members of the struct using the '->' operator

Probably need to check for NULL before dereferencing or accessing members.

For large structs, this is more efficient if we need to pass it as an argument for example.

Pointers and structs

```
Point p;
Point q;
p.x = 1;
p.y = 2;
q = p;
printf("%d", q.y);
```

Copying structs needs a little care to be taken….

This is a ‘**shallow**’ copy using the simple assignment operator.
For many structs this is fine, however, as soon as other, more complex data members (i.e. with ‘deep’ content) are used – then this wouldn’t work.

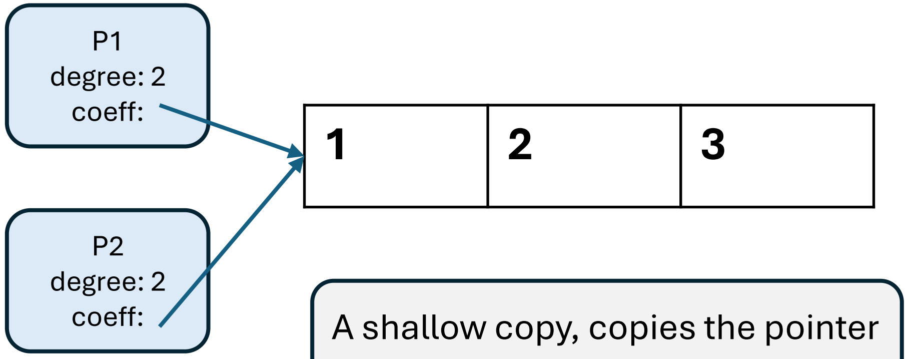
Pointers and structs (Shallow Copy)

```
typedef struct OrderCoeff{

    int degree;
    int *coeff;   //Dynamically allocated array

} polynomial;
```

Say we had a struct like this, with a pointer as a member.

The pointer is a member of the struct – not the array that it points to.

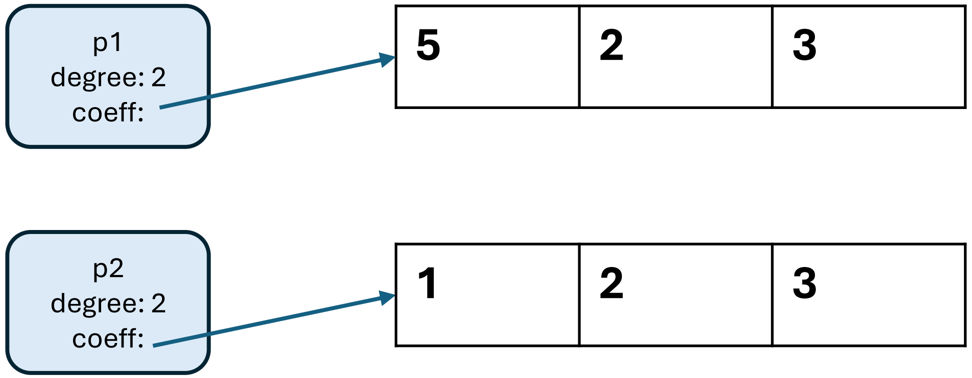
```
int main(int argc, char **argv)
{
    polynomial p1;
    p1.degree = 2;
    p1.coeff = malloc(3 * sizeof(int));  //allocate space for the dynamic array

    for(int i=0; i<3;i++){               //fill the dynamic array for p1
        p1.coeff[i] = i+1;
    }

    polynomial p2;
    p2 = p1;
    p1.coeff[0] = 5;

    printf("%d", p2.coeff[0]);

}
```



A shallow copy, copies the pointer members of the structs only

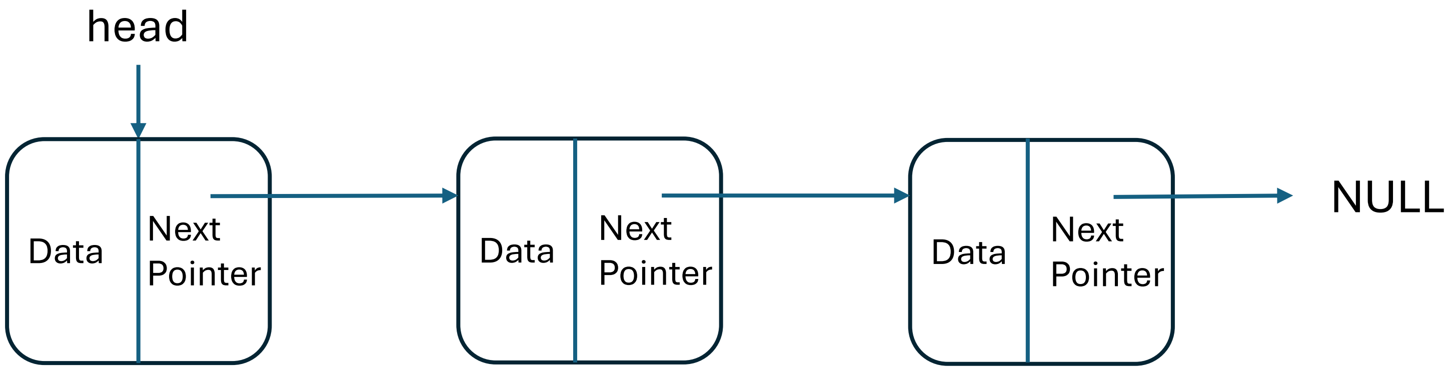
Pointers and structs (Deep Copy)

```
polynomial p1;
p1.degree = 2;
p1.coeff = malloc(3 * sizeof(int));  //allocate space for the dynamic array in p1

for(int i=0; i<3;i++){                //fill the dynamic array for p1
    p1.coeff[i] = i+1;
}

polynomial p2;
p2.coeff = malloc(3 * sizeof(int));
p2.degree = p1.degree;
for(int i=0; i<3;i++){                //deep copy the dynamic array for p2 from p1
    p2.coeff[i] = p1.coeff[i];
}

p1.coeff[0] = 5;
printf("%d", p2.coeff[0]);
```



Pointers and Structs:
A singly Linked List

- A linked list is a fundamental collection data structure which has a number of advantages over an array
  - It is more efficient for inserting or deleting an item from the collection



Pointers and Structs:
A singly Linked List

```c
typedef struct Node{
    int data;
    struct Node* nextPointer;
}Node;
```

```c
void outputList(Node* n){
    while(n){
        printf("%d -> ", n->data);
        n = n->nextPointer;
    }
}
```

```c
int main(int argc, char **argv)
{
    //Create the memory space for the new node
    Node* firstNode = (Node*)malloc(sizeof(Node));
    firstNode->data = 5;

    //Create the memory space for the new node
    Node* secondNode = (Node*)malloc(sizeof(Node));
    secondNode->data = 8;

    //Create the memory space for the new node
    Node* thirdNode = (Node*)malloc(sizeof(Node));
    thirdNode->data = 2;

    //Join the nodes together in a list
    firstNode->nextPointer=secondNode;
    secondNode->nextPointer=thirdNode;
    thirdNode->nextPointer=NULL;

    //output list
    Node* temp = firstNode;
    outputList(temp);

    return 0;
}
```

Other linked lists need a slightly different implementation with structs:

Doubly linked list
Circular linked list

In conclusion …

- In this session, we have covered:
  - Struct in C
    - Concept
    - Defining, using
    - Accessing using pointers
    - Shallow copy/Deep copy of structs
  - Structs and Linked Lists
    - Concept
    - Using structs to create linked lists