C Programming - Deck 17

Pointers and Structures

Prof. Jyotiprakash Mishra
mail@jyotiprakash.org

Pointers to Structures

- Pointer can point to a structure variable
- Syntax: `struct Point *ptr;`
- Access members using arrow operator: `ptr->x`
- Arrow operator `->` is shorthand for `(*ptr).x`
- Dot operator `.` for structure variable
- Arrow operator `->` for structure pointer
- Passing pointers to functions is more efficient
- Avoids copying entire structure

Program 1: Basic Pointer to Structure

```c
#include <stdio.h>
struct Point {
  int x;
  int y;
};
int main() {
  struct Point p1 = {10, 20};
  struct Point *ptr = &p1;
  printf("Using dot: (%d, %d)\n", p1.x, p1.y);
  printf("Using arrow: (%d, %d)\n",
    ptr->x, ptr->y);
  printf("Using *: (%d, %d)\n",
    (*ptr).x, (*ptr).y);
  return 0;
}
```

Output:

```
Using dot: (10, 20)
Using arrow: (10, 20)
Using *: (10, 20)
```

Three ways to access: . -¿ and (*).

Program 2: Modifying Structure Through Pointer

```c
#include <stdio.h>
struct Point {
  int x;
  int y;
};
int main() {
  struct Point p1 = {5, 10};
  struct Point *ptr = &p1;
  printf("Before: (%d, %d)\n", p1.x, p1.y);
  ptr->x = 15;
  ptr->y = 25;
  printf("After: (%d, %d)\n", p1.x, p1.y);
  return 0;
}
```

Output:

```
Before: (5, 10)
After: (15, 25)
```

Modifying structure members via pointer

Program 3: Array of Structure Pointers

```c
#include <stdio.h>
struct Student {
  int roll;
  int marks;
};
int main() {
  struct Student s1 = {1, 85};
  struct Student s2 = {2, 90};
  struct Student s3 = {3, 78};
  struct Student *arr[3] = {&s1, &s2, &s3};
  int i;
  for (i = 0; i < 3; i++) {
    printf("Roll: %d, Marks: %d\n",
      arr[i]->roll, arr[i]->marks);
  }
  return 0;
}
```

Output:

```
Roll: 1, Marks: 85
Roll: 2, Marks: 90
Roll: 3, Marks: 78
```

Array of pointers to structures

Program 4: Passing Structure Pointer to Function

```c
#include <stdio.h>
struct Point {
  int x;
  int y;
};
void display(struct Point *p) {
  printf("Point: (%d, %d)\n", p->x, p->y);
}
int main() {
  struct Point p1 = {10, 20};
  display(&p1);
  return 0;
}
```

Output:

```
Point: (10, 20)
```

Efficient: passes address, not copy

Program 5: Modifying Structure in Function

```c
#include <stdio.h>
struct Point {
  int x;
  int y;
};
void shift(struct Point *p, int dx, int dy) {
  p->x += dx;
  p->y += dy;
}
int main() {
  struct Point p1 = {10, 20};
  printf("Before: (%d, %d)\n", p1.x, p1.y);
  shift(&p1, 5, 10);
  printf("After: (%d, %d)\n", p1.x, p1.y);
  return 0;
}
```

Output:

```
Before: (10, 20)
After: (15, 30)
```

Function modifies original structure

Program 6: Returning Structure Pointer

```c
#include <stdio.h>
struct Point {
  int x;
  int y;
};
struct Point p1 = {10, 20};
struct Point p2 = {30, 40};
struct Point* getMax(struct Point *a,
  struct Point *b) {
  if (a->x + a->y > b->x + b->y)
    return a;
  return b;
}
int main() {
  struct Point *max = getMax(&p1, &p2);
  printf("Max: (%d, %d)\n", max->x, max->y);
  return 0;
}
```

Output:

```
Max: (30, 40)
```

Returning pointer to existing structure

Program 7: Structure with Nested Pointer Members

```c
#include <stdio.h>
struct Person {
  char *name;
  int age;
};
int main() {
  struct Person p1 = {"Alice", 25};
  struct Person *ptr = &p1;
  printf("Name: %s\n", ptr->name);
  printf("Age: %d\n", ptr->age);
  ptr->name = "Bob";
  ptr->age = 30;
  printf("Updated: %s, %d\n",
    ptr->name, ptr->age);
  return 0;
}
```

Output:

```
Name: Alice
Age: 25
Updated: Bob, 30
```

Structure containing pointer member

Program 8: Pointer to Array of Structures

```c
#include <stdio.h>
struct Point {
  int x;
  int y;
};
int main() {
  struct Point arr[3] = {{1,2}, {3,4}, {5,6}};
  struct Point *ptr = arr;
  int i;
  for (i = 0; i < 3; i++) {
    printf("Point %d: (%d, %d)\n",
      i, (ptr+i)->x, (ptr+i)->y);
  }
  return 0;
}
```

Output:

```
Point 0: (1, 2)
Point 1: (3, 4)
Point 2: (5, 6)
```

Pointer arithmetic with structures

Program 9: Compare Structures Using Pointers

```c
#include <stdio.h>
struct Point {
  int x;
  int y;
};
int equal(struct Point *p1,
  struct Point *p2) {
  return (p1->x == p2->x && p1->y == p2->y);
}
int main() {
  struct Point a = {10, 20};
  struct Point b = {10, 20};
  if (equal(&a, &b)) {
    printf("Points are equal\n");
  } else {
    printf("Points are not equal\n");
  }
  return 0;
}
```

Output:

```
Points are equal
```

Comparing structure members

Program 10: Distance Between Two Points

```c
#include <stdio.h>
#include <math.h>
struct Point {
  int x;
  int y;
};
double distance(struct Point *p1,
  struct Point *p2) {
  int dx = p2->x - p1->x;
  int dy = p2->y - p1->y;
  return sqrt(dx*dx + dy*dy);
}
int main() {
  struct Point a = {0, 0};
  struct Point b = {3, 4};
  printf("Distance: %.2f\n", distance(&a, &b));
  return 0;
}
```

Output:

```
Distance: 5.00
```

Computing distance using pointers

Program 11: Self-Referential Structure

```c
#include <stdio.h>
struct Node {
  int data;
  struct Node *next;
};
int main() {
  struct Node n1 = {10, NULL};
  struct Node n2 = {20, NULL};
  struct Node n3 = {30, NULL};
  n1.next = &n2;
  n2.next = &n3;
  struct Node *ptr = &n1;
  while (ptr != NULL) {
    printf("%d -> ", ptr->data);
    ptr = ptr->next;
  }
  printf("NULL\n");
  return 0;
}
```

Output:

```
10 -> 20 -> 30 -> NULL
```

Linked list concept with pointers

Program 12: Finding Student with Highest Marks

```c
#include <stdio.h>
struct Student {
  char name[20];
  int marks;
};
int main() {
  struct Student s[3] = {
    {"Alice", 85},
    {"Bob", 92},
    {"Charlie", 78}
  };
  struct Student *max = &s[0];
  int i;
  for (i = 1; i < 3; i++) {
    if (s[i].marks > max->marks) {
      max = &s[i];
    }
  }
  printf("Highest: %s (%d)\n",
    max->name, max->marks);
  return 0;
}
```

Output:

```
Highest: Bob (92)
```

Pointer to max element in array

Program 13: Swap Two Structures Using Pointers

```c
#include <stdio.h>
struct Point {
  int x;
  int y;
};
void swap(struct Point *p1,
  struct Point *p2) {
  struct Point temp = *p1;
  *p1 = *p2;
  *p2 = temp;
}
int main() {
  struct Point a = {10, 20};
  struct Point b = {30, 40};
  printf("Before: (%d,%d) (%d,%d)\n",
    a.x, a.y, b.x, b.y);
  swap(&a, &b);
  printf("After: (%d,%d) (%d,%d)\n",
    a.x, a.y, b.x, b.y);
  return 0;
}
```

Output:

```
Before: (10,20) (30,40)
After: (30,40) (10,20)
```

Swapping entire structures

Program 14: Nested Structure with Pointers

```c
#include <stdio.h>
struct Date {
  int day;
  int month;
  int year;
};
struct Employee {
  char name[20];
  struct Date dob;
};
int main() {
  struct Employee e = {"John", {15, 8, 1990}};
  struct Employee *ptr = &e;
  printf("Name: %s\n", ptr->name);
  printf("DOB: %d/%d/%d\n",
    ptr->dob.day, ptr->dob.month, ptr->dob.year);
  return 0;
}
```

Output:

```
Name: John
DOB: 15/8/1990
```

Accessing nested structure members

Program 15: Calculate Average Marks Using Pointers

```c
#include <stdio.h>
struct Student {
  int roll;
  int marks;
};
float average(struct Student *arr, int n) {
  int sum = 0;
  int i;
  for (i = 0; i < n; i++) {
    sum += (arr + i)->marks;
  }
  return (float)sum / n;
}
int main() {
  struct Student s[3] = {{1,85},{2,90},{3,78}};
  printf("Average: %.2f\n", average(s, 3));
  return 0;
}
```

Output:

```
Average: 84.33
```

Processing array of structures

Program 16: Pointer to Structure with Array Member

```c
#include <stdio.h>
struct Student {
  char name[20];
  int marks[3];
};
int main() {
  struct Student s = {"Alice", {85, 90, 78}};
  struct Student *ptr = &s;
  int i;
  printf("Name: %s\n", ptr->name);
  printf("Marks: ");
  for (i = 0; i < 3; i++) {
    printf("%d ", ptr->marks[i]);
  }
  printf("\n");
  return 0;
}
```

Output:

```
Name: Alice
Marks: 85 90 78
```

Structure with array accessed via pointer

Program 17: Update Structure Array Using Pointer

```c
#include <stdio.h>
struct Point {
  int x;
  int y;
};
void translate(struct Point *arr, int n,
  int dx, int dy) {
  int i;
  for (i = 0; i < n; i++) {
    (arr + i)->x += dx;
    (arr + i)->y += dy;
  }
}
int main() {
  struct Point p[2] = {{1,2}, {3,4}};
  translate(p, 2, 10, 20);
  printf("(%d,%d) (%d,%d)\n",
    p[0].x, p[0].y, p[1].x, p[1].y);
  return 0;
}
```

Output:

```
(11,22) (13,24)
```

Modifying all structures in array

Program 18: Size of Structure and Pointer

```c
#include <stdio.h>
struct Student {
  int roll;
  char name[20];
  float marks;
};
int main() {
  struct Student s;
  struct Student *ptr;
  printf("Size of structure: %lu\n",
    sizeof(s));
  printf("Size of pointer: %lu\n",
    sizeof(ptr));
  printf("Advantage of pointer: less copy\n");
  return 0;
}
```

Output:

```
Size of structure: 28
Size of pointer: 8
Advantage of pointer: less copy
```

Pointer is much smaller than structure

Program 19: Sort Structures Using Pointers

```c
#include <stdio.h>
struct Student {
  int roll;
  int marks;
};
int main() {
  struct Student s[3] = {{3,78},{1,85},{2,90}};
  struct Student *p[3] = {&s[0],&s[1],&s[2]};
  struct Student *temp;
  int i, j;
  for (i = 0; i < 2; i++) {
    for (j = i+1; j < 3; j++) {
      if (p[i]->roll > p[j]->roll) {
        temp = p[i];
        p[i] = p[j];
        p[j] = temp;
      }
    }
  }
  for (i=0; i<3; i++)
    printf("Roll:%d Marks:%d\n",
      p[i]->roll, p[i]->marks);
  return 0;
}
```

Output:

```
Roll:1 Marks:85
Roll:2 Marks:90
Roll:3 Marks:78
```

Sorting pointers, not structures

Program 20: Complex Structure with Multiple Pointers

```c
#include <stdio.h>
struct Node {
  int data;
  struct Node *prev;
  struct Node *next;
};
int main() {
  struct Node n1 = {10, NULL, NULL};
  struct Node n2 = {20, NULL, NULL};
  n1.next = &n2;
  n2.prev = &n1;
  struct Node *ptr = &n1;
  printf("Forward: %d -> %d\n",
    ptr->data, ptr->next->data);
  ptr = &n2;
  printf("Backward: %d -> %d\n",
    ptr->data, ptr->prev->data);
  return 0;
}
```

Output:

```
Forward: 10 -> 20
Backward: 20 -> 10
```

Doubly linked list concept

Key Takeaways

- Pointer to structure: `struct Type *ptr;`
- Arrow operator `->` for pointer: `ptr->member`
- Dot operator `.` for variable: `var.member`
- `ptr->member` equivalent to `(*ptr).member`
- Passing pointers avoids copying entire structure
- Array of structure pointers for flexible data
- Self-referential structures enable linked lists
- Pointer arithmetic works with structure arrays
- Understanding this is crucial for data structures