ESC101N
Fundamentals of Computing

Arnab Bhattacharya
arnabb@iitk.ac.in

Indian Institute of Technology, Kanpur
http://www.iitk.ac.in/esc101/

1st semester, 2010-11

Tue, Wed, Fri 0800-0900 at L7

Structures

- *Structures* are customized data types
- It is declared using the keyword struct

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

- struct Point is a structure having two variables x and y
- Variables in a structure are called members
- A variable of the type structure can be defined using

```
struct Point p;
```

Structures

- *Structures* are customized data types
- It is declared using the keyword struct

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

- struct Point is a structure having two variables x and y
- Variables in a structure are called members
- A variable of the type structure can be defined using

```
struct Point p;
```

- A structure type can be explicitly defined using typedef

```
typedef struct Point point;
```

- point becomes an alias for struct Point
- A structure variable can then simply be defined as

```
point s;
```

Members

- Structures can be initialized during declaration

```
point p = {4.0, -3.0};
```

- By default, they are initialized to 0 (or '\0')
  - Same as array
- Its members can be explicitly assigned values
- `.` notation to access members

  `structure_variable.member_name`

```
p.x = 4.0;
p.y = -3.0;
```

- Members behave just like ordinary variables
- Size of a structure is the combined size of its members
- Example: Size of `point` is $8 + 8 = 16$ bytes

Functions returning structures

- Since structures are variables, a function can return them

```
point copy_point(point s)
{
  point p;
  p.x = s.x;
  p.y = s.y;
  return p;
}
```

- This can also be used to create structures

```
q = f(9.0, -3.0);
```

Functions returning structures

- Since structures are variables, a function can return them

```
point copy_point(point s)
{
  point p;
  p.x = s.x;
  p.y = s.y;
  return p;
}
```

- This can also be used to create structures

```
q = f(9.0, -3.0);
```

- Copying can also be done simply by

```
q = p;
```

- A structure is just a variable
  - Different from array

Functions returning structures

- Since structures are variables, a function can return them

```
point copy_point(point s)
{
  point p;
  p.x = s.x;
  p.y = s.y;
  return p;
}
```

- This can also be used to create structures

```
q = f(9.0, -3.0);
```

- Copying can also be done simply by

```
q = p;
```

- A structure is just a variable
  - Different from array
- Structures cannot be compared

```
if (q == p) // error
```

Passing structures to functions

- Since structures are variables, they can be passed to functions
- Modifying the elements of a structure inside a function is *temporary*

```
void modify(point p, double c, double d)
{
  p.x = c;
  p.y = d;
}
```

- The following code prints 5.0 and −3.0

```
point q = {5.0, -3.0};
modify(q, 9.0, 1.0);
printf(‘‘%lf %lf\n’’, q.x, q.y);
```

Pointers to structures

- A pointer to a structure can be defined

```
point *ptr, p;
ptr = &p;
```

- When a pointer to structure is passed to a function, modifying the elements of the structure inside the function becomes *permanent*

```
void modify(point *p, double c, double d)
{
  p->x = c;
  p->y = d;
}
```

- The following code prints 9.0 and 1.0

```
point q = {5.0, -3.0};
modify(&q, 9.0, 1.0);
printf(‘‘%lf %lf\n’’, q.x, q.y);
```

Pointers to structures

- A pointer to a structure can be defined

```
point *ptr, p;
ptr = &p;
```

- When a pointer to structure is passed to a function, modifying the elements of the structure inside the function becomes *permanent*

```
void modify(point *p, double c, double d)
{
  p->x = c;
  p->y = d;
}
```

- The following code prints 9.0 and 1.0

```
point q = {5.0, -3.0};
modify(&q, 9.0, 1.0);
printf(‘‘%lf %lf\n’’, q.x, q.y);
```

- `->` notation to access members using pointers
  `structure_pointer->member_name`
- `ptr->x` is same as `(*ptr).x`

Structure operations I

```c
#include <stdio.h>
#include <math.h>

struct Point
{
    double x;
    double y;
};  // defining a structure

typedef struct Point point; // defining a new type using structure

point new_point(double c, double d) // structure as return value
{
    point p;

    p.x = c;
    p.y = d;

    return p;
}

double distance(point a, point b)   // structure as parameter
{
    double d = 0.0;

    d = sqrt(pow(a.x - b.x, 2) + pow(a.y - b.y, 2));

    return d;
}
```

Structure operations II

```
void modify_wrong(point p, double c, double d)
{
    p.x = c;    // modifying members inside function is temporary
    p.y = d;
}

void modify_pointer(point *p, double c, double d)
{
    p->x = c;   // modifying members using structure pointer is permanent
    p->y = d;   // -> notation
}

int main()
{
    struct Point p, q;  // declaring using structure
    point s;    // declaring using type
    point t = {9.0, -5.0};  // initializing during declaration
    double d;
    point *ptr;

    printf("%lf %lf\n", p.x, p.y);  // by default, values are 0

    q.x = 4.0;  // accessing or modifying the members in a structure
    q.y = -3.0; // . notation

    d = distance(p, q);
    printf("Distance = %lf\n", d);

    //p = {9.0, -5.0};  // error
```

Structure operations III

```
    //printf("%lf %lf\n", p.x, p.y);

    p = new_point(7.0, -1.0);
    printf("%lf %lf\n", p.x, p.y);

    modify_wrong(q, 7.0, -1.0);
    printf("%lf %lf\n", q.x, q.y);

    ptr = &p;
    modify_pointer(ptr, 2.0, -5.0);
    printf("%lf %lf\t%lf %lf\n", p.x, p.y, ptr->x, ptr->y);

    printf("Size of p is %d, size of ptr is %d\n", sizeof(p), sizeof(ptr));

    //if (q == p)   // error
    //  printf("Equal structures\n");
}
```

Nested structures

- A structure can have another structure as its member

```
typedef struct Line
{
  point p;
  point q;
} line;
```

- Note: `typedef` definitions can be combined – equivalent to

```
struct Line
{
  point p;
  point q;
};
typedef struct Line line;
```

- Value `x` of `point p` of variable `l` of type `line` can be accessed as: `l.p.x`
- The `.` operator has left-to-right associativity

Array of structures

- An array of structures can be simply defined as

  ```
  point t[3];
  ```

- Each individual structure is accessed as `t[0]`, etc.
- A member of a structure is accessed as `t[i].x`, etc.
- All operations allowed on normal arrays are allowed on array of structures
- It is equivalent to a pointer to structure

Array of structures

```c
#include <stdio.h>

typedef struct Point
{
    double x;
    double y;
} point;

int main()
{
    point t[3];
    int i;

    for (i = 0; i < 3; i++)
    {
        t[i].x = i;
        t[i].y = 2 * i;
        printf("%lf %lf\n", t[i].x, t[i].y);
    }
}
```

Pointer in a structure

- A structure can have a pointer as its member

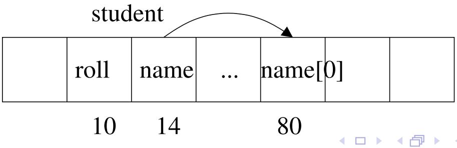
```
typedef struct Student
{
  int roll;
  char *name;
} student;
```

- Declaring a variable of type student just declares the pointer name – it does not allocate space for it

```
student s;
strcat(s.name, ‘‘.’’); // error
```

- Memory for name has to be allocated explicitly using malloc

```
s.name = (char *)malloc(30 * sizeof(char));
```



Pointer in a structure

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

typedef struct Student
{
    int roll;
    char *name;
} student;

int main()
{
    student s;

    s.name = (char *)malloc(30 * sizeof(char));

    scanf("%d%s", &s.roll, s.name);

    printf("%d %s\n", s.roll, s.name);

    strcat(s.name, "A");

    printf("%d %s\n", s.roll, s.name);

    free(s.name);
}
```