Structures

R. Inkulu
http://www.iitg.ac.in/rinkulu/

Outline

Motivational examples

modular code: keeping relevant data (and operations over it) at one place

- student/employee data
- customer details
- characteristics of a geometric entity
- sparse matrix representation
- polynomial representation
- etc.,

Motivation with an example

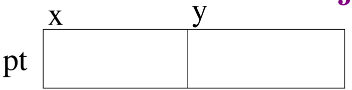
```
struct point {
   double x;   //a member
   int y;      //a member
};

int main(void) {
   struct point pt;
      //defines object pt of type point
    pt.x = 25.68; pt.y = 35;
      //initializing point stru members
    printf("%lf, %d", pt.x, pt.y);
      //prints 25.68, 35
}
```

- *structure* is a custom-type to group one or more objects of not necessarily of the same type

    whereas array comprises of homogeneous typed objects

- helps in developing modular code

Memory layout of a structure object




```
struct point {
   double x;
   int y;
};
int main(void) {
   struct point pt;
      //defines object pt of type point
    printf("%d, %d, %d",
           sizeof(pt.x), sizeof(pt.y), sizeof(pt));
      //prints 8, 4, 12
    printf("%p, %p, %p", &pt.x, &pt.y, &pt);
      //prints 0xbfb40290, 0xbfb40298, 0xbfb40290
}
```

- linear memory layout
- always qualify member name with the object

Initializing members while defining an object

```
struct point {
   double x;
   int y;
};

int main(void) {
    struct point pt = {25.68, 30};
       //defines and initializes
    printf("%lf, %d", pt.x, pt.y);
       //prints 25.68, 30
}
```

Bit-by-bit copy of objects

```
struct point {
   double x;
   int y;
};

int main(void) {
   struct point pt1, pt2;
   pt1.x = 20.23; pt1.y = (int) pt1.x+80;
   pt2 = pt1;
   printf("%lf, %d", pt2.x, pt2.y);
      //prints 20.23, 100
}
```

Aliasing with a new type name

```
struct point {
      double x;
      int y; };

int main(void) {
   typedef struct point Point;
      //now Point is a synonym for struct point
   Point pt;
   pt.x = 20.23; pt.y = (int) pt.x+80;
   printf("%lf, %d", pt.x, pt.y);
      //prints 20.23, 100
}
```

- *typedef* does not introduce a new type instead aliases a new name for an existing type

  this helps in giving module-specific name to a type; more importantly, in writing portable code across multiple platforms

Type equivalence

```
struct point1 { double d; int i; };
typedef struct point1 P1;
typedef struct point1 P2, P3;
struct point2 { double d; int i; };
typedef struct point2 P4;
```

- *Name type equivalence*: Each type declaration introduces a new type, distinct from all others. (In the example, P1, P2, P3, P4, struct point1, and struct point2 are distinct from each other. Adv: easier to incorporate by the compiler)
- *Structural type equivalence*: Two types are same only if they have identically ordered members in the corresponding structures. (In the example, P1, P2, P3, P4, struct point1, and struct point2 are all equivalent. Disadv: harder for the compiler to enforce)
- *Decalaration type equivalence*: Two types are equivalent only if they lead back to the same structure type. (In the example, P1, P2, P3, and struct point1 types are equivalent whereas P4 and struct point 2 are equivalent but distinct from the rest)

C uses structural type equivalence for all types except for structures and unions, for which C uses declaration equivalence.

Typical structure type declarations and object defintions

- struct T $\{\ldots\}$;

  declares T as a new type

  define an object o by writing *struct* T o

- struct $\{\ldots\}$ o;

  defines o as an object

- struct T $\{\ldots\}$;
  typedef struct T T';

  creates T' as an alias for the type *struct* T

- typedef struct $\{\ldots\}$ T; //commonly used

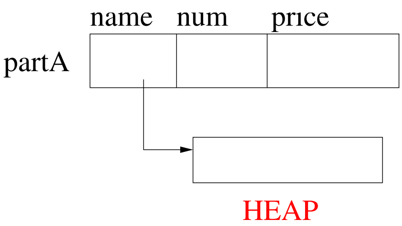
  creates an anonymous structure and aliases it with T

  define an object o of type T by writing T o

Structure with array members

```
typedef struct {
   char name[50];
   int num;
   double price;
} Part;

int main(void) {
   Part partA;
   strcpy(partA.name, "Crank");
   partA.num = 1000;
   partA.price = 48.20;
   printf("%s, %c, %lf", partA.name, partA.name[2],
    partA.price);
      //prints Crank, a, 48.20
}
```

Structure with pointer members




```
typedef struct {
   char *name;
   int num;
   double price;
} Part;
int main(void) {
   Part partA;
   partA.name = (char*) malloc(count*sizeof(char));
   strcpy(partA.name, "abcd"); partA.num = 1000;
   ...
   free(partA.name);  }
```

Array of structures

parts[0]	parts[1]	...	parts[19]

```
typedef struct {
   char *name;
   int num;
   double price;
} Part;

int main(void) {
    Part parts[20];
    printf("%d, %d\n", sizeof(Part), sizeof(parts));
    //prints 16, 320
}
```

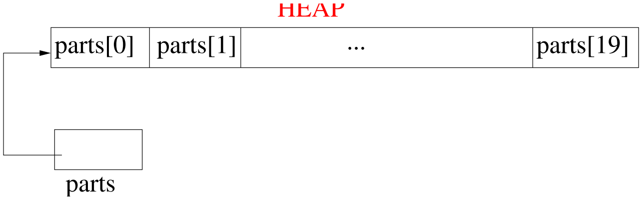
Array of structures

```
int main(void) {
   Part parts[20];
   ...
   const char *buf = "abcdefgh";
   parts[10].name =
       (char*) malloc((strlen(buf)+1)*sizeof(char));
   strcpy(parts[10].name, buf);
   parts[10].num = 1000; parts[10].price = 48.20;

   printf("%s, %lf", parts[10].name, parts[10].price);
      //prints abcdefgh, 48.20000

   printf("%d, %d", sizeof(parts[12]), sizeof(parts));
      //prints 16, 320
   ...
   free(parts[10].name);
}
```

Initializing array of structures while defining

```
struct key {
    const char *word;
    int count;
};

int main(void) {
    typedef struct key K;
    K keytab[] = { {"auto", 123}, {"break", 345} };
    ...
    printf("%d, %s\n", keytab[0].count, keytab[1].word);
      //prints 123, break
}
```

Pointer to a contiguous sequence of structure objects



(for count equals to 20)

```
typedef struct {
   char *name;
   int num;
   double price;
} Part;
int main(void) {
   ...  //count value computed
   Part *parts= (Part*) malloc(count*sizeof(Part));
   ...
   free(parts); }
```

Pointer to a contiguous sequence of structure objects (cont)

```
int main(void) {
   const char *buf = "abcdefgh";
   Part *parts=
      (Part*) malloc(numParts*sizeof(Part));
   ...
   parts[10].name =
             (char*) malloc((strlen(buf)+1)*sizeof(char));
   strcpy(parts[10].name, buf);
   parts[10].num = 1000;
   parts[10].price = 48.20;
   printf("%s, %lf", parts[10].name, parts[10].price);
      //prints Crank, 48.20000
   ...
   free(parts[10].name);
   ...
   free(parts);
   ...  }
```

Pointer to contiguous sequence of structures (cont)

```
int main(void) {
   const char *buf = "abcdefgh";
   Part *parts=
         (Part*) malloc(numTypesOfParts*sizeof(Part));
   ...
   (parts+10)->name =
         (char*) malloc((strlen(buf)+1)*sizeof(char));
   strcpy(parts[10].name, buf);
   (*(parts+10)).num = 1000;
   parts[10].price = 48.20;
   printf("%s, %lf", parts[10].name, (parts+10)->price);
   ...
   free((parts+10)->name); free(parts); }
```

- -> is the *dereference operator* to access a structure member through a pointer;

  parts[10].num is same as (*(parts+10)).num, or (parts+10)->num;
  $(*(parts+0)).num$ is written as parts->num

A structure containing other structure objects

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

typedef struct { Point p[3]; } Cube;

typedef struct { Cube *c; } Hypercube;
```

```
...
Hypercube hc;
hc.c = (Cube*) malloc(numdim*sizeof(Cube));
        //assume numdim >= 2
...
(hc.c+1)->p[2].y = 45;
printf("%lf", (hc.c+1)->p[2].y);
...
free(hc.p);
```

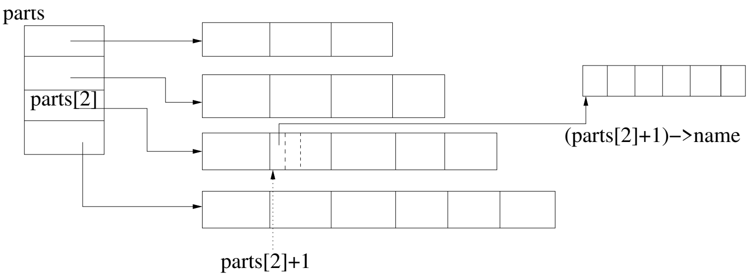
Self-referential structures

```
struct List{
  char data;
  struct List *link;
};

struct List abc;
abc.data = 'c';
abc.link = NULL;
printf("%c, %p", abc.data, abc.link);
  //prints c, (nil)
```

- a member of a structure pointing to an object of the same structure type: unique in the sense that before the compiler learns the structure, it encounters a pointer to the same [1]

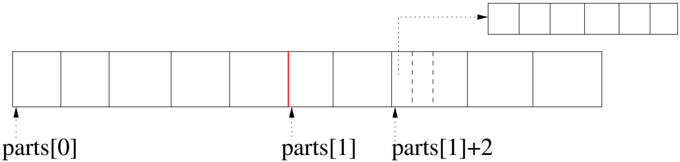
[1] obviously, structure A cannot have an object of type A as a member of A

Array of pointers to varying number of structure objects




```
Part *parts[4];
for (int i=0; i<4; i++)
    parts[i] = (Part*) malloc((3+i)*sizeof(Part));
...
parts[2][1].name = (char*) malloc(count*sizeof(char));
...
free((parts[2]+1)->name);
for (int i=0; i<4; i++) free(parts[i]);
```

Array of pointers to fixed number of structure objects




```
Part (*parts)[5];
parts = (Part (*)[5]) malloc(count*sizeof(Part[5]));
...
parts[1][2].name = (char*) malloc(count*sizeof(char));
...
free(parts[1][2].name);
free(parts);
```

review

- structures with members as
  - primitive objects (like int, double, etc.,)
  - arrays
  - pointers to primitive type objects
  - members which are objects of other structures
  - pointers to objects of other structures
- array of structure objects
- self-referential structures
- pointer to a contiguous sequence of structure objects
- array of pointers to varying number of structure objects
- array of pointers to fixed number of structure objects

Outline

1. Introduction
2. Functions and structure objects
3. Unions
4. Self-alignment

Passing structure objects to functions

semantically incorrect Code

```
typedef struct { double x;  double y; } Point;
void swap(Point a, Point b) {
  Point tmp = a;
  a = b;
  b = tmp;
}

int main(void) {
  Point p1={10, 20}, p2={40, 50};
  swap(p1, p2);
  printf("%lf, %lf", p1.x, p1.y);
    //prints 10.00000, 20.00000  !
}
```

- structures are passed by value i.e., object contents are copied bit-by-bit

Passing structure objects to functions

```
typedef struct { double x;  double y; } Point;
void swap(Point *a, Point *b) {
  Point tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(void) {
  Point p1={10, 20}, p2={40, 50};
  swap(&p1, &p2);
  printf("%lf, %lf", p1.x, p1.y);
    //prints 40.00000, 50.00000
}
```

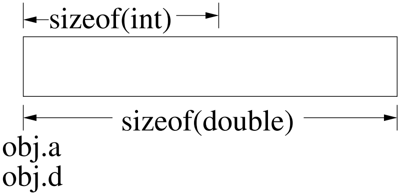
Returning structure objects from functions

```
typedef struct { double x;  double y; } Point;
Point constructPtObj(double xcoor, double ycoor) {
   Point obj;
   obj.x = xcoor;  obj.y = ycoor;
   return obj;
}
int main(void) {
  Point a = constructPtObj(25, 30);
  printf("%lf, %lf", a.x, a.y);
    //prints 25.00000, 30.00000
}
```

- automatic object *obj* in constructPtObj is copied bit-by-bit to an object named *a* in main
- functions are allowed to return pointers to structures — again, protocol for freeing the memory need to be addressed

Outline

Size of union




```
typedef union{
  int a;
  double d;
} TwoNums;
TwoNums obj;
printf("%d", sizeof(obj));
   //prints 8
```

- a single variable that can legitimately hold any one of several types from the member list
- size of a union object equals to the member that occupies the maximum space

Accessing a union object

```
int main(void) {
union {
  int a;
  double d;
} obj;
printf("%p, %p \n", &obj.a, &obj.d);
   //prints 0xbfa4e5b8, 0xbfa4e5b8
obj.a = 20;  obj.d = 39.45;
printf("%d, %lf \n", obj.a, obj.d);
   //prints -1717986918, 39.450000
}
```

- holds at different time different types of objects — same as structure except that only one member makes sense at any one time
- it is the programmer's responsibility to keep track of which type is currently stored in a union

Typical way union objects are used

```
struct ABC {
   enum {INT=1, DOUBLE} type;
   union {
     int a;
     double d;
   } uobj;
};
struct ABC x;
x.type = INT; x.uobj.a = 5;
...
switch(x.type) {
  //based on the value of x.type,
  //access the member of uobj
  case INT:
     printf("%d", x.uobj.a); break;
  case DOUBLE:
     printf("%lf", x.uobj.d); break;
}
```

Custom-typed objects within union

```
typedef struct { double dia; double depth;} PartAInfo;
typedef struct { int len; double depth;} PartBInfo;
typedef struct {
    enum {PartAType=1, PartBType} objtype;
    union { PartAInfo partA; PartBInfo partB; } obj;
} PartInfo;

PartInfo p;
p.objtype = PartBType;
p.obj.partB.len = 5;
...
switch (p.objtype) {
   case PartAType:
      ...  break;
   case PartBType:
      printf("%d\n", p.obj.partB.len); ...  break;
}
```

Outline

Self-alignment of primitive types

```
int i; short s; char c; long l;

printf("%d, %d, %d, %d, %d\n", sizeof(i), sizeof(s),
 sizeof(c), sizeof(l));
//prints 4, 2, 1, 8

printf("%p, %p, %p, %p, %p\n", &i, &s, &c, &l);
//prints 0xbfdf07cc, 0xbfdf07ca, 0xbfdf07c9, 0xbfdf07d8
```

- *self-aligned*: value v of a primitive type $typeA$ is stored starting from only bytes whose address is an non-negative integer multiple of $sizeof(typeA)$
- modern processor architectures' are designed to access addresses that are self-aligned efficiently; hence, compilers generate binary code to exploit the same

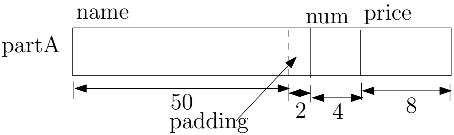
Self-alignment of structures

```
typedef struct {
   char *name;
   int num;
   double price;
} Part;

int main(void) {
   Part partA;
   printf("%d, %p\n", sizeof(partA), &partA);
     //prints 16, 0xbf88bf10
}
```

- like primitive typed values, custom objects are also self-aligned: object o of a custom type (structure) T can only be stored starting from bytes whose address is a non-negative integer multiple of the most restrictive member of T[2]

[2] helps in calculating internal and trailing paddings (see below)

Internal padding for self-alignment

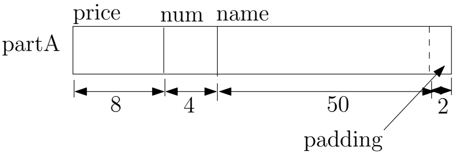



```
typedef struct {
   char name[50];
   int num;
   double price; } Part;
```

```
int main(void) {
    Part partA;
    printf("%d, %d, %d\n", sizeof(Part), (void*)&partA.num-(void*)partA.name,
     (void*)&partA.price-(void*)&partA.num); }
```

- padding is need for the purpose of member self-alignment (considering the self-alignment of the objects instantiated from that structure)

Trailing padding for self-alignment




```
typedef struct {
double price;
int num;
char name[50]; } Part;
```

```
int main(void) {
    Part partA;
    printf("%d, %d, %d\n", sizeof(Part), (void*)&partA.num-(void*)&partA.price,
     (void*)partA.name-(void*)&partA.num);
}
```

- trailing padding is needed to take care of defining array of structure objects
- however, leading padding is not allowed (according to the standard, address of first member of a structure object must need to be same as the address of object itself)

Examples

- typedef struct{ short, char} A;
- typedef struct{ char*, int} B;
- typedef struct{ char*, short, int } C;
- typedef struct{ int, char, short} D;
- typedef struct{ int, char [3], short [10]} E;
- typedef struct { C, E [6], D [10]} F; [3]
- try more . . .

homework: analyze the sizes and draw the memory layouts

[3] recursively apply the self-alignment rules