Structures and Enumerated Types

CS2023 Winter 2004

Outcomes: Structures

- *C for Java Programmers*, Chapters 11 (11.1 to 11.4) and 12 (12.1 to 12.2),
- After the conclusion of this section you should be able to
  - Declare, intialize, and use structures and pointers to structures
  - Efficiently pass structures to functions
  - Combine structures and arrays to create arrays of structures and structures containing arrays
  - Use enumerated types when you need ordered collections of named constants

Structures

Structures are user defined data types, which represent *heterogeneous* collections of data.

```
struct info {
      char firstName[20];
      char lastName[20];
      int age;
};
struct info i1, i2;
info j;
```

Structures

```
struct info {
    char firstName[20];
    char lastName[20];
    int age;
} i1, i2;
typedef struct info { /*can be omitted */
    char firstName[20];
    char lastName[20];
    int age;
 } InfoT;
```

Structure tag

Using Structures

```
typedef struct info {
      char firstName[20];
      char lastName[20];
      int age;
 } InfoT;

 InfoT p1;
```

In order to *access* members of a structure:

```
   p1.age = 18;
   printf("%s\n", p1.firstName);
```

Structure Errors

```
struct example { ... };
example e;
struct example e;

struct example { ... }  /* no ; */
```

Nested Structures

```
typedef struct {
      char firstName[20];
      char lastName[20];
      int age;
 } InfoT;
 typedef struct {
      InfoT info;
      double salary;
 } EmployeeT;
 EmployeeT e1;
 e1.info.age = 21;
```

Assignments & Comparing Structures

```
InfoT i1, i2;
i1 = i2;

struct { int a[10]} a1, a2;
a1 = a2 /*legal, since a1, a2 are
         structures*/

i1 == i2
strcmp(i1.firstName, i2.firstName) == 0
  && strcmp(i1.lastName, i2.lastName) ==
  0 && i1.age == i2.age
```

Structures & Pointers

```
struct pair {
        double x;
        double y;
} w, *p;
typedef struct pair {
        double x;
        double y;
} PairT, *PairTP;
PartT x;
PairTP p;
```

Structures & Pointers

Memory for pointers to structures must be initialized in same way as for other pointers:

- using address of another structure
- using dynamic memory allocation

```
typedef struct pair {
     double x;
     double y;
} PairT, *PairTP;
PairT w;
PairTP p = &w;
```

Structures & Pointers

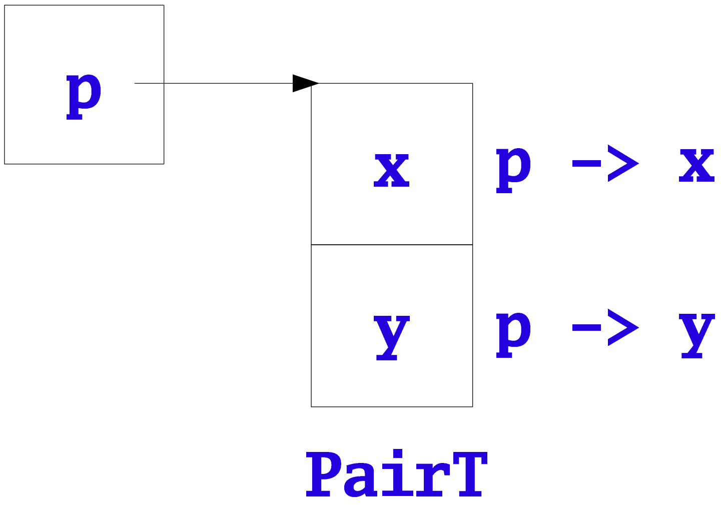
```
typedef struct pair {
     double x;
     double y;
} PairT, *PairTP;
PairTP q;
if((q = malloc(sizeof(PairT))) == NULL) …\
/* or */
if((q = malloc(sizeof(struct pair))) == NULL) …

q->x = 1;      (*q).x = 1;    *q.x = 1;
q->y = 3.5;
```

Structures & Pointers

If **p** is a pointer to a structure that has a member **w**, then

p->w

gives access to **w**.



Size of a Structure

You cannot assume that the size of a structure is the same as the sum of the sizes of all its members, because the compiler may use padding to satisfy memory alignment requirements.

```
struct ci{
  char a;
  int b;
} s;
```

How large is **`s`**?

- Say int = 4 bytes and char = 1 byte, sizeof(s) is not necessarily = 5, since some O/S require that data begin on some multiple number of bytes (typically 4). So **`a`** could be followed by a three-byte hole. Can also have holes at end of a structure.

Structures and Functions

```
typedef struct pair {
      double x;
      double y;
} PairT, *PairTP;
PairT constructorFunc(double x, double y) {
   PairT p;

   p.x = x;
   p.y = y;
   return p;
}
PairT w = constructorFunc(1, 2.2); /* COPY */
```

Structures and Functions

- Previous function inefficient because structure is created on function's stack, then copied to calling function's stack
- Use call by reference instead

```
void constructorP(PairTP this,
                  double x, double y) {
   this->x = x;
   this->y = y;
}

PairT w;
PairTP p;

constructorP(&w, 1, 2); /* copy only doubles */

constructorP(p, 1, 2);
if((p = malloc(sizeof(PairT))) == NULL) error;
constructorP(p, 1, 2);
```

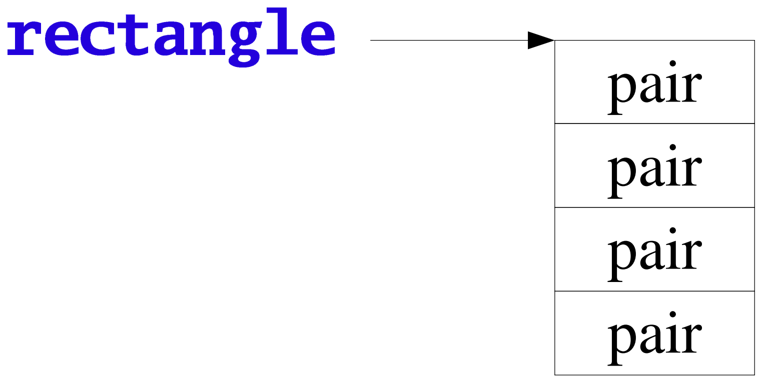
```
PairTP constructor(double x, double y) {
/* client responsible for deallocation */
   PairTP p;
   if((p = malloc(sizeof(PairT))) == NULL)
      return NULL;
   p->x = x;
   p->y = y;
   return p;
}

int compare(const PairTP p, const PairTP q) {
   return p->x == q->x && p->y == q->y;
}
```

```
PairTP p1 = constructor(1, 2);
PairTP p2 = constructor(1, 3);

int i = compare(p1, p2);

free(p1);
free(p2);
```

Avoid leaving garbage:

```
i = compare(p1, constructor(3.5, 7));
```

Blocks of Structures

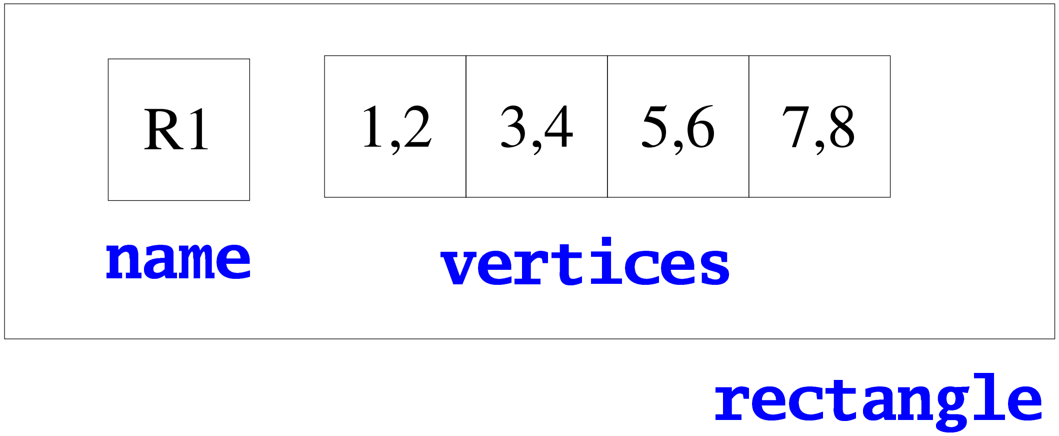


```
PairTP rectangle;
PairTP aux;
double x, y;
if((rectangle=
  malloc(4*sizeof(PairT)))==NULL)error;
 for(aux = rectangle; aux < rectangle + 4; aux++) {
   printf("Enter two double values:");
   if(scanf("%lf%lf", &x, &y) != 2) /* error */
     break;
   constructorP(aux, x, y);
 }
```

Blocks of Structures

```
int i;
for (i = 0; i < 4; i++)
  printf("vertex %d = (%f % f)\n",
         i, rectangle[i].x, rectangle[i].y);
```

Structures and Arrays



R1

1,2 3,4 5,6 7,8

name

vertices

rectangle

```
#define MAX 20
typedef struct {
   char name[MAX+1];
   PairT vertices[4];
} RectangleT, *RectangleTP;
RectangleT rectangle;
```

```
void Show(const RectangleTP s) {
   int i;
   printf("Rectangle %s\n", s->name);
   for(i = 0; i < 4; i++)
      printf("vertex %d = (%f %f)\n", i,
       s->vertices[i].x, s->vertices[i].y);
}
```

`s->name`	the array of characters
`s->vertices`	the array of pairs
`s->vertices[i]`	the `i`-th pair
`s->vertices[i].x`	the `x`-coordinate of the `i`-th pair

Initialization of Structures

```
typedef struct {
     double x;
     double y;
} PairT;
PairT q = {2.3, 4};
typedef struct {
     char name[MAX+1];
     PairT vertices[4];
} RectangleT;
RectangleT s = { "first",
           { {0, 1}, {2, 3}, {4, 5},
           {1, 2} }};
```

Example: Quadratic Equation

```
typedef struct quadratic {
  double a;
  double b;
  double c;
  double root[2];
} QuadT;

int roots(QuadT *q);

/*
 * Solve a quadratic equation for real roots,
  returning 1.
 * If the roots are imaginary or a == 0 return 0.
 */
```

Example: Quadratic Equation

```
int roots(QuadT *q)
{
  double r = q->b * q->b - 4.0 * q->a * q->c;

  if ( q->a == 0 || r < 0.0)
    return 0;

  r = sqrt(r);
  q->root[0] = (-q->b + r)/(2.0 * q->a);
  q->root[1] = (-q->b - r)/(2.0 * q->a);
  return(1);
}
```

Example: Quadratic Equation

```
main ()
{
  QuadT eqn;

  printf("Please enter a b and c\n");
  scanf("%lf %lf %lf", &eqn.a, &eqn.b, &eqn.c);
  if (roots(&eqn) == 1)
    printf("Roots are %f %f\n", eqn.root[0],
  eqn.root[1]);
  else
    printf("Roots are imaginary\n");
}
```

Enumerated Types

Enumerated types are ordered collections of named constants; e.g.

```
enum opcodes {
   lvalue, rvalue, push, plus
};
typedef enum opcodes {
   lvalue, rvalue, push, plus
} OpcodesT;
enum opcodes e;
OpcodesT f;
```

Enumerated Types

The definition of

```
enum opcodes {
   lvalue, rvalue, push, plus
};
```

introduces four constants: **lvalue**, **rvalue**, **push**, and **plus**; all are of type convertible to **int**:

lvalue	represents the integer value	0
rvalue		1

and so on

Enumerated Types

A declaration of an enumerated type may also *explicitly* define values:

```
enum opcodes {
   lvalue = 1, rvalue, push, plus
};
   enum opcodes e;
   e = lvalue;
   if(e == push) …

   int i = (int)rvalue; /* equal to 2 */
```

Enumerated Types

To represent function return codes; e.g.

failure because a file can not be opened

failure because a file can not be closed

success

```
typedef enum {
   FOPEN, FCLOSE, FOK
} FoperT;
```

Enumerated Types

Consider a function

```
 FoperT process();
```

To output the result of calling this function as a string

```
char *Messages[] = {
      "File can not be opened",
      "File can not be closed",
      "Successful operation",
      "This can not happen"
};
printf("result of calling process() is %s\n",
   Messages[(int)process()];
```