CS2002
Computer Systems
Lecture 5

Structs and Unions

Jon Lewis (JC 0.26)

School of Computer Science

University of St Andrews

Overview

- Structures in C
  - declaration
  - initialisation
  - structures in expressions
  - passing to and returning from functions
  - pointers to structs
  - Pre-processor and header gaurds
- Modular Code design
  - Some Person ADT examples
- Unions
  - defining, using

Structures

- Structure declarations introduce a type of several fields
- Superficially similar to classes in Java
- A structure is a logical choice for storing a collection of related data items.

Structure Types

- Ways to name a ‘type’ of structure (Declare a “structure tag”)
- The declaration of a **structure tag** named `Part`:

```
struct Part {
   int number;
   char name[NAME_LEN+1];
   int on_hand;
}; // <- Notice this semi-colon!
...
struct Part part1, part2; //define vars
```

- Use `typedef` to define a type name for **`struct`** `Part`

```
    typedef struct Part Part;
```

  - Can precede struct Part definition
  - Useful when defining recursive structs (e.g. liked list type - we will see later)

Structure Types (2)

- Or all-in-one definition of a type named `Part`:

```
typedef struct Part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} Part;
```

- Either way, after the typedef
  - `Part` can be used in the same way as the built-in types:

```
Part part1, part2;
```

Nested Structures

- Structure types can contain other structure types

```
typedef struct Point    { // normally put in Point.h
  int x;
  int y;
} Point;

typedef struct Line { // normally put in Line.h
  Point from;
  Point to;
} Line;


typedef struct Polygon { // normally put in Polygon.h
  Point nodes[MAX_NODES];
} Polygon;
```

Need for Header Guards

```
#ifndef POINT_H_
#define POINT_H_
 typedef struct Point {
      int x;
      int y;
} Point;
#endif
```

- imagine a source file file.c #including Point.h and Line.h (where Line.h also included Point.h)
- Header guard avoids compilation of file.c including Point.h **twice** (which would cause an error due to multiple definitions of the Point struct)

Structs are closest thing in C to Java classes

- Structs can only contain members.
  - Members can be other structs.
- Structs do not have any other features of classes
  - Member functions
  - Static member functions
  - Private members
  - Inheritance
  - Constructor / Destructor.
- However
  - You can make all these things with plain C functions if you want them.
  - Examples later on

Initialising Structure Values

- This nests for both nested arrays, and nested structures

```
Line base = {{0,0}, {10,20}};

Line p[2] = { {{0,0},{10,20}}, {{10,10},{20,30}} };
```

Initialising Structures (2)

- Same as arrays, if you give no initalizer, fields are initiated to 0 for globals, random data for local variables.
- If you define at least one element, all others are set to 0!

```
Line p[2] = {0}; // same as
Line p[2] = {{{0}}}; // same as
Line p[2] = {{{0,0},{0,0}},{{0,0},{0,0}}};
```

Operations on Structures

- The period in this context is actually a C operator.
- It takes precedence over most other operators:

```
    scanf("%d", &part1.on_hand);
```

`&` computes the address of `(part1.on_hand)`.

- The other major structure operation is assignment:

```
    part2 = part1;
```

  - The effect of this statement is to copy `part1.number` into `part2.number`, `part1.name` into `part2.name`, and so on.

Accessing Structures

- Access with a '.' notation, like Java's

```
// Assume point & line are declared and typedef’ed.
Point p = {1,2};
printf("(%d,%d)", p.x, p.y);

Line l = {{1,1},{2,2}};
l.from.x = l.to.x;
l.from.y = l.to.y;

typedef struct { point points[10];} Ten_points;
Ten_points ten;
ten.points[2].x = 0;
```

Operations on Structures

- Arrays can't be copied using the = operator, but

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

- Some programmers exploit this property by creating "dummy" structures to enclose arrays that will be copied later.

- ==, !=, etc. are not valid, you have to write your own functionality to compare individual struct members

Function and Structures

- Functions can accept and return structs:

```
Point Point_difference(Point p1, Point p2) {
  p1.x -= p2.x;
  p1.y -= p2.y;
  return p1;
}
```

Structs are passed to functions, **and returned**, by value (copied)

or

```
Point Point_difference(Point p1, Point p2) {
  Point retval = { p1.x - p2.x, p1.y - p2.y };
  return retval;
}
```

Arrays and Structs

- Arrays are passed by reference, so changing them in a function changes the original.
- An array inside a struct is passed/copied by value.

```
struct S { int i[10]; int j;};

// Changes original i. Does not change j.
void addoneA(int i[], int j) {
  i[0] += 1; j += 1;
}
// Does not change original s.i or s.j
void addoneS(struct S s) {
  s.i[0] += 1; s.j += 1;
}
// Changes both original a[0].i[0] and a[0].j
void addoneSA(struct S a[]) {
  a[0].i[0] +=1; a[0].j +=1;
}
```

f() to Initialise Structures

- Structures can be initialised by a function (obviously)

```
void Point_printDetails(Point this); // in other file or below

Point new_Point(int x, int y) {
  Point this;
  this.x = x;
  this.y = y;
  return this;
}

Point emptyPoint() {
  return new_Point(0, 0);
}

int main() {
  Point p1 = new_Point(1, 2);
  Point_printDetails(p1);
}
```

Pointers to structs

- **Struct**s can have pointers to them in the same way as **int**, **double**, etc.

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


// This will change the original p
void Point_shiftX(Point* p, int diff) {
  (*p).x += diff;
}


// Same as above using p-> instead of (*p).
void Point_shiftY(Point* p, int diff) {
  p->x += diff;
}
```

EXAMPLES

Person examples on studres for Lecture 05

UNIONS

Unions

- A *union***,** like a structure, consists of one or more members, **possibly of different types**.

- The compiler allocates **only enough space** for the largest of the members, which overlay each other within this space.

- Assigning a new value to one member **alters** the values of the other members as well.

Unions vs Structs

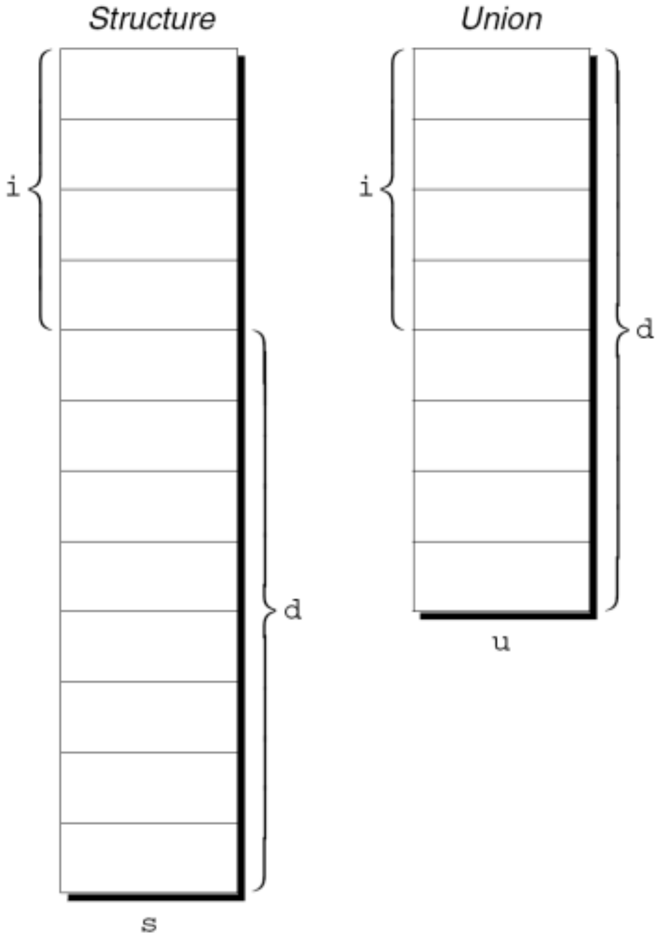
- An example of a union variable:

```
union u {
   int i;
   double d;
};
```

- The declaration of a union closely resembles a structure declaration:

```
struct s{
   int i;
   double d;
};
```

Where they differ:



Unions

- unions are superficially similar to structs, but behave very differently.
- All items in a union use the same memory. Writing one overwrites all the other values in the union.

Using Unions to Build Mixed Data Structures

- Suppose that we need an array whose elements are a mixture of `int` and `double` values.

```
typedef union {
   int i;
   double d;
} Number;

Number number_array[1000];
// array elements can be int OR double!


number_array[0].i = 5;
number_array[1].d = 8.395;
```

Members are Indistinguishable

- There’s no easy way to tell which member of a union was last changed and therefore contains a meaningful value.

- Consider:

```
void print_number(Number n) {
  if (n contains an integer) // C has no such feature!
    printf("%d", n.i);
  else
    printf("%g", n.d);
}
```

Add a "Tag Field" to a struct containing union

- Redefine `Number` as a struct with an embedded union:

```
#define INT_KIND 0
#define DOUBLE_KIND 1

typedef struct {
  short tag;    /* tag field */
  union { int i; double d; } u;
} Number;
```

- Always couple assignments to u with corresponding tag:

```
n.tag = INT_KIND;
n.u.i = 82;
```

Add a “Tag Field” to a struct containing union

- A function that takes advantage of this capability:

```
void print_number(Number n){
  if (n.tag == INT_KIND)
    printf("%d", n.u.i);
  else
    printf("%g", n.u.d);
}
```