京都先端科学大学

KYOTO UNIVERSITY OF ADVANCED SCIENCE

京都発世界人財

Information Processing 2　11

structures
arrays of structures, pointers to structures
typical structure applications

Ian Piumarta

Faculty of Engineering, KUAS

last week: more arrays and pointers

arrays of strings
sorting lines of text from the input
modified `qsort()` that sorts strings
`day_of_year()` and `month_day()`
`month_name()` and initialising array of strings
`echo` program
find command line string in text lines
`sort` using function pointers
  to parameterise behaviour
many complex declarations
`dcl` converts declarations to English
`undcl` converts English to declaration

this week: structures

point structure
rect structure
point and rect functions
pointers to structures and (*x).y vs. x->y
structure for dynamic strings
arrays of structures
structure for counting words
selection sort

Chapter 6. Structures
6.1 Basics of Structures
6.2 Structures and Functions
6.3 Arrays of Structures
6.4 Pointers to Structures

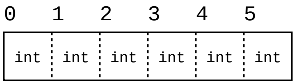
structures are collections of related variables

a *structure* is a collection of related variables that are managed as a group

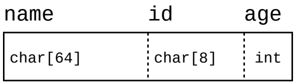
- the values always occur together
- typically, multiple attributes of a single, complex entity

compared to arrays:

array



structure



group of related values
elements are **numbered**
elements have **same** type
variable size

group of related values
elements are **named**
elements can have **different** types
fixed size

structures are collections of related variables

student: name, id, credits, gpa

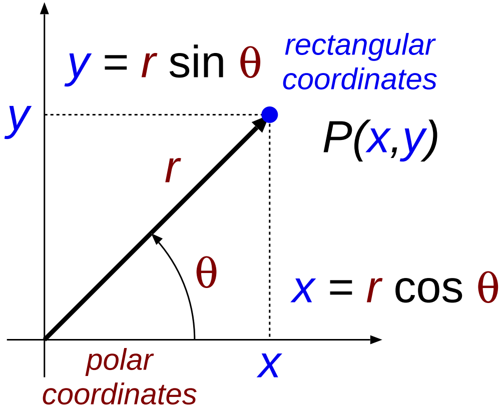
```
char    name[64]; // John Smith
char    id[8];    // 2019m999
int     credits;  // 42
float   gpa;      // 3.9
```

polar point: r and a (angle, or 'theta' θ)

```
double r, theta;
```

Cartesian point: x and y

```
double x, y;
```



a structure is a type, grouping several members into one object

a Cartesian point has two attributes: x and y

declare them as variables and then group them into a single structure like this:

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

this defines a new type called `struct point`

the name `point` is called the structure *tag*

the variables inside the structure are called its *members*

the `struct` groups its members so they can be treated as a single value

structures can be named and referred to later using their tag

each `struct` declaration defines a new type: `struct` *tag*

you can declare variables of that new type

```
struct point { double x, y; };    // declare a structure type
struct point p, q, r;             // declare three 'struct point's
```

you can combine the type and variable declarations

```
struct point { double x, y; } p, q;    // declare type and two vars
struct point r;                        // declare another variable
```

structures are initialised like arrays, but can also be copied

structures are initialised by writing the value of each member in curly braces, in order

```
struct point pt = { 1920, 1080 };
```

the members of a structure are accessed as: *structure-variable* . *member-name*

```
printf("%f,%f", pt.x, pt.y);    // x and y members of pt
```

assigning one structure to another *copies* the entire structure

```
struct point pt = { 1920, 1080 }, qt = { 1024, 768 };
pt = qt;    // overwrite pt with contents of qt
```

structures are also *copied* when passed to, or returned from, functions

```
pt = point_doubled(qt);
struct point point_doubled(struct point p) {
  struct point q = { p.x + p.x, p.y + p.y };
  return q;
}
```

structures are initialised like arrays, but can also be copied

you cannot compare two structures using a relation

- you can write your own function(s) to do that

```
int point_leftOf(struct point p, struct point q) {
  return p.x < q.y;
}

int point_below(struct point p, struct point q) {
  return p.y < q.y;
}
```

members can be treated just like any other variable

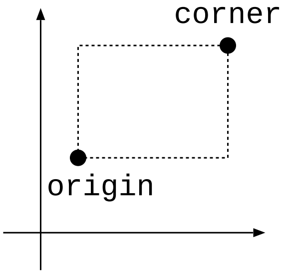
```
struct point point_add(struct point p, struct point q) {
  p.x += q.x;  // increment x and y in local copy of first argument
  p.y += q.y;
  return p;    // return copy of local point p
}

struct point rt = point_add(pt, qt);
```

a structure can be a member of another structure

structures can be nested

a rectangle can be made from two points

```
struct rectangle {
  struct point origin;
  struct point corner;
};
```



initialisation of nested structures can be done separately or all at once

```
struct point p = { 1, 2 }, q = { 3, 4 };

struct rectangle s = { p, q };  // initialise nested structures

struct rectangle r = {
  { 1, 2 },  // origin point    // initialise the members all at once
  { 5, 5 }   // corner point    // with extra {...} for every structure
};
```

structures can be passed to, and returned from, functions

common pattern: constructor function for a structure type

```
struct point point_new(int x, int y) {
  struct point p = { x, y };
  return p;
}
```

```
struct rectangle rectangle_new(struct point o, struct point c) {
  struct rect r = { o, c };
  return r;
}
```

structure parameters are passed by value (they are copied)

passing a structure to a function is like passing all the members individually

- not efficient!

sometimes it is better to pass a pointer to the structure

structure pointers are just like pointers to any other variable

```
void point_print(struct point *pp) {
  printf("{ %f, %f }", (*pp).x, (*pp).y);
}
```

note: the parentheses are *necessary* as `*pp.x` means `*(pp.x)` which is illegal

- because `pp` is a *pointer* — *not* a structure

accessing a member through a pointer has special notation

the pattern (**structure-pointer*) . *member-name* is very common

a shorthand is available: *structure-pointer* -> *member-name*

the above example can be rewritten

```
void point_print(struct point *pp) {
  printf("{ %f,%f }", pp->x, pp->y);
}
```

note that both . and -> associate from left to right

```
struct rectangle r, *rp = &r;
```

```
r.origin.x    ==   (r.origin).x
rp->origin.x ==   (rp->origin).x
```

structures let us implement ‘missing’ data types, such as string

let’s implement a `struct string` to represent dynamic strings

- store the length explicitly
- no need for nul terminator character
- can store character 0 as a ‘normal’ character

```
struct string
{
  char *chars;  // storage for the string, allocated with malloc()
  int   size;   // length of the string
};
```

operations

- create a string from a C string
- print a string
- append a character to the end
- concatenate a string to the end

storage for the string will be managed automatically

constructor a `struct string` from C string

create a string from a C string

```
struct string string_new(char *chars) {
  struct string s = { strdup(chars), strlen(chars) };
  return s;
}
```

this function returns a structure that should be copied into a string variable

```
struct string s = string_new("hello");
```

when the contents of a string are no longer needed they should be freed

```
void string_free(struct string *s) {
  free(s->chars);
  s->chars = 0;
  s->size  = 0;
}
```

passing pointers to structures avoids copying many members

print a string, given a pointer (its location), using two formatting tricks

- in '%s' conversion, the precision limits the number of characters printed
  - `printf("%.5s", s)` prints no more than 5 characters from `s`
- giving '*' as the precision reads the precision from the next `int` argument
  - `printf("%.*s", n, s)` prints no more than `n` characters from `s`

```
void string_print(struct string *s) {
  printf("\"%.*s\"", s->size, s->chars);
}

void string_println(struct string *s) {
  string_print(s);
  printf("\n");
}
```

reallocate storage automatically when more space is needed

note: `ptr = strdup(s)` copies `s` into new storage

note: `ptr = realloc(ptr, n)` resizes storage to `n` bytes

- the original storage *must* be obtained from `malloc()` or `strdup()`

```
char *s = strdup("hello"); // s contains 6 bytes of storage
s = realloc(s, 13);        // increase s to 13 bytes
strcat(s, ", world");      // append ", world"
```

append a character to a string

```
struct string *string_append(struct string *s, int c) {
  s->chars = realloc(s->chars, s->size + 1);  // grow chars by 1 byte
  s->chars[s->size++] = c;                    // store c in new byte
  return s;
}
```

next week: recursive structures, bit fields, unions

Chapter 6. Structures
6.5 Self-referential Structures
6.6 Table Lookup
6.7 Typedef
6.8 Unions
6.9 Bit-fields

`sizeof()` and counting number of array elements
pointer version of `findWord()`
data structure: binary tree or linked list
using `typedef` for nodes
unions and bit fields

assignment

please download assignment from

MS Team “Information Processing 2”, “General” channel

exercises

structure for points

rectangular point properties

printing points

polar points

polar point property: radius

appending to string structures

extending strings

better string extend function

13:00	5	last week: more arrays and pointers
13:05	5	this week: structures
13:10	5	structures are collections of related variables
13:15	5	a structure is a type, grouping several members into one object
13:20	5	structures can be named and referred to later using their tag
13:25	5	structures are initialised like arrays, but can also be copied
13:30	5	members are variables, accessed by their name
13:35	5	a structure can be a member of another structure
13:40	5	structure parameters are passed by value (they are copied)
13:45	5	accessing a member through a pointer has special notation
13:50	5	arrays of structures replace parallel arrays of related data
13:55	5	example: read a word from the input
14:00	5	replace parallel arrays with array of structures
14:05	5	structures let us implement ‘missing’ data types, such as string
14:10	5	a constructor from C strings is useful
14:15	5	passing pointers to structures avoids copying many members
14:20	5	storage can be reallocated to make more space, opaquely
14:25	5	next week: recursive structures, bit fields, unions
14:30	10	*break*
14:40	90	exercises
16:10	00	end