Notes 14.0: Structures and unions

COMP9021 Principles of Programming

School of Computer Science and Engineering
The University of New South Wales

2014 session 1

Introduction

- Simple variables and arrays offer only limited means for representing and organizing data.
- Structures enable to represent more complex data in a flexible and natural way: they permit a group of related variables to be treated as a unit instead of as separate entities.
- More precisely, a structure is a collection of variables, possibly of different types, grouped together under a single name.
- For instance, a point in a 2-dimensional space can be represented by a structure of the form:

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


Declarations (1)

- The keyword `struct` introduces the structure declaration, which is a list of declarations enclosed in braces.
- In the previous example, `point` is a structure tag; it can be used subsequently as a shorthand for the structure declaration.
- A structure tag is optional, but it is necessary if the structure template is to be used more than once.
- In the previous example, `x` and `y` are the structure members.
- Structure members are local to the structure declaration, hence can have the same names as structure members of other structures or as ordinary variables.
- A structure declaration defines a complex type.
- We can follow a structure declaration with a list of variables; this sets aside space for them.

Declarations (2)

- For instance, we can write

```
struct {int x;  int y;} pt1, pt2, pt3;
```

  or

```
struct point {int x;  int y;} pt1, pt2, pt3;
```

  to declare `pt1`, `pt2` and `pt3` and set aside space for them.
- A structure declaration that is not followed by a list of variables only describes a template for the structure.
- Still, the tag in a tagged structure declaration that is not followed by a list of variables can be used later in definitions of instances of the structure.
- For instance, given the structure definition on slide 2, we can write

```
struct point pt1, pt2, pt3;
```

  This also declares `pt1`, `pt2` and `pt3`, and sets aside space for them.

Structure initialization

- A variable of complex type can be initialized by following its declaration with a list of expressions, one for each member of the structure.
- These expressions must match the types of the corresponding members, and they must be constant if the structure has static storage duration.
- For instance, a point of coordinates $(320, 200)$ can be defined by:

```
struct point pt = {320, 200};
```

- It is also possible to initialize an automatic variable of complex type T by assigning to it a variable of type T, as with

```
struct point pt = {320, 200};
struct point pt_copy = pt;
```

or by assigning to it the value returned by a function of type T (to be considered later).

Structure members (1)

- We can refer to a member of a particular structure thanks to the dot operator, in a construction of the form

structure_name.structure_member

- For instance, to print the coordinates of the point `pt` we can write:

```
printf("%d, %d\n", pt.x, pt.y);
```

- For another example, to compute the distance between `pt` and the origin, we can cast the point coordinates to `double`s and write:

```
double dist;
dist = sqrt((double)pt.x * pt.x +
            (double)pt.y * pt.y);
```


Structure members (2)

- Structures can be nested. For instance, a line segment can be represented by the structure

```
struct segment {
    struct point left_pt;
    struct point right_pt;
};
```

- If we declare `seg` as

```
struct segment seg;
```

then

```
seg.left_point.x
```

refers to the x coordinate of the left endpoint of the line segment `seg`.

Example

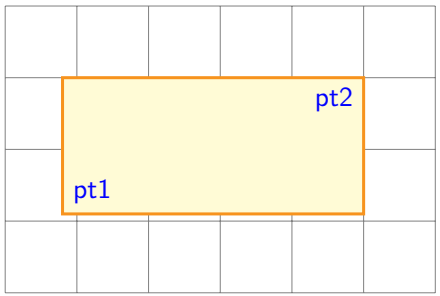
The program book.c uses two built-in functions:

- `fgets()`, used for reading any input, that takes as arguments:
  - the address of the string that will store the input;
  - the maximum size of the input plus 1 (to store a terminating null character indicating the end of the string);
  - a file pointer identifying the file to be read (`stdin` is the address of standard input).
- `strchr()`, that returns the address of the first occurrence of the character provided as second argument in the string provided as first argument (or the null pointer if there is no such occurrence).

Functions of complex type (1)

Consider the following structure declaration.

```
struct rect {
    struct point pt1;
    struct point pt2;
};
```



Functions of complex type (2)

```
struct point make_point(int x, int y) {
    struct point temp;
    temp.x = x;
    temp.y = y;
    return temp;
}
```

- `make_point()` makes a point from x and y coordinates: it returns a value of type `point`.
- There is no conflict between argument names and the name of the members of the structure; actually the reuse of variables stresses the relationship.

Functions of complex type (3)

`make_point()` can be used to initialize automatic variables:

- either initializing the structure members of a variable of type `rect`, or
- initializing a variable of type `point`:

```
struct rect screen;
screen.pt1 = make_point(0, 0);
screen.pt2 = makepoint(XMAX, YMAX);

struct point middle =
  make_point((screen.pt1.x + screen.pt2.x) / 2,
             (screen.pt1.y + screen.pt2.y) / 2);
```


Functions of complex type (4)

- Functions can also take complex types as arguments, whether they are or not of complex type.
- `add_point()` is an example of the first kind;
- `point_is_in_rect()`, which checks whether a point is inside a rectangle, is an example of the second type.

```
struct point add_point(struct point p1, struct point p2) {
    p1.x += p2.x;
    p1.y += p2.y;
    return p1;
}
bool point_is_in_rect(struct point p, struct rect r) {
    return p.x >= r.pt1.x && p.x <= r.pt2.x
        && p.y >= r.pt1.y && p.y <= r.pt2.y;
}
```

Pointers to structures (1)

- If a large structure is to be passed to a function, it is more efficient to pass a pointer than to copy the whole structure, especially if the structure is large.
- Structure pointers are like pointers to ordinary variables: the declaration

```
struct point *pp;
```

  says that `pp` is a pointer to a structure of type `struct point`.
- If `pp` points to a structure of type `struct point` then:
  - `*pp` is the structure;
  - `(*pp).x` and `(*pp).y` are its members, where the parentheses are necessary because `.` has higher precedence than `*`.
- `pp->x` is a shorthand for `(*pp).x`. More generally, if `p` is a pointer to a structure and `memb` is a member of this structure then `p->memb` is a shorthand for `(*p).memb`.

Pointers to structures (2)

- Both `.` and `->` associate from left to right.
- Given `struct rect r, *rp = &r;`
  these four expressions are equivalent:

```
r.pt1.x
rp->pt1.x
(r.pt1).x
(rp->pt1).x
```

- Together with `()` for function calls and `[]` for subscripts, `.` and `->` are at the top of the precedence hierarchy and thus bind very tightly.
- For instance, if `p` is a pointer to a structure one of whose members is a variable `len` of type `int` then

```
++p->len
```

  is equivalent to `++(p->len)`, hence increments `len`.

Operations on structures

- The only legal operations on a structure are:
  - assigning to it;
  - accessing its members;
  - taking its address with `&`.

  When a structure is assigned to another, each member of the former is assigned to the corresponding member of the latter (compare with arrays: it is not possible to assign an array to another).
  Assignments are illustrated by programs names1.c, names2.c and names3.c.
- To manipulate a structure we can either:
  - pass the entire structure;
  - pass the structure members separately;
  - pass a pointer to the structure.

  The program funds.c illustrates the three ways of manipulating a structure.

Arrays of structures

- Arrays of structures are like any other kind of array.
- For instance, `struct book copies[MAX_BOOKS]`
  declares `copies` to be an array of size `MAX_BOOKS` whose members are of type `struct book`.
- The program books.c is an extension of book.c that allows to read multiple entries.

Note that in contrast to arrays, the name of a structure does not point to the address of the structure, as illustrated in friends.c

typedef (1)

- `typedef` enables to create a name for a type. It should be compared with `#define`:
  - `#define` can be used to give names to any expression, whereas `typedef` is limited to giving symbolic names to types;
  - `#define` is interpreted by the preprocessor whereas `typedef` is interpreted by the compiler;
  - still `typedef` is more flexible than `define`.
- For instance, to create a name for `unsigned char`s we can write either:

```
typedef unsigned char Byte;
```

or

```
#define Byte unsigned char
```

either of which can then be used to declare variables of type `Byte` or derived from `Byte`:

```
Byte x, y[10], *z;
```


typedef (2)

- On the other hand,

```
typedef char *String;
```

cannot be replaced by

```
#define String char *
```

Indeed, in the latter case,

```
String string1, string2
```

would make `string1` a pointer to a `char` but `string2` a `char`.
- `typedef` are useful with structures. *E.g.*:

```
typedef struct complex {
    float real;
    float imag;
} Complex;
```

- Then `Complex` can be used instead of `struct complex` to represent complex numbers: this is a more convenient and recognizable name for a type that is commonly used.

typedef (3)

- Tags can be omitted when using `typedef`:

```
typedef struct {double x; double y;} Rect;
Rect r1 = {3., 6.};
Rect r2;
r2 = r1;
```

which is translated into:

```
struct {double x; double y;} r1 = {3., 6.};
struct {double x; double y;} r2;
r2 = r1;
```

- When 2 structures match in member names and types, these two structures are considered to be of the same type, which makes

```
r2 = r1;
```

a valid operation.

Unions

- Unions are similar to structures, in that their declaration defines a template listing their elements, accessible using the `.` or `->` operators.
- Unions are declared with the `union` keyword:

```
union temp {
    char   name[20];
    double x, y;
    int    i;
}
```

- A particular instance of a union can only hold a single member (with the previous example, *either* an array of `char`s *or* a double, whose name can be *either* `x` *or* `y`, *or* an `int`).
- When a union is declared, the compiler determines the largest size of the union member. This size is by definition the size of the union (which can be computed using the `sizeof` operator).

The program union.c illustrates.

Using unions (1)

Unions can facilitate the creation of generic functions able to process data of different types. *E.g.*:

```
struct point {int x, y;};
struct line {int x1, y1, x2, y2;};
struct circle {int x, y, radius;};
struct triangle {int x1, y1, x2, y2, x3, y3;};
struct object {
    int object_type;
    union {
        struct point p;
        struct line l;
        struct circle c;
        struct triangle t;
    };
};
```


Using unions (2)

```
void display(struct object x) {
    if (x.object_type == POINT)
        display_point(&x);
    if (x.object_type == LINE)
        display_line(&x);
    ...
}
```