Structures and Union

Jinkyu Jeong
(jinkyu@skku.edu)
Review

- **Bitwise operations**
  - You need them for performance in terms of space and time
  - Shifts are equivalent to arithmetic

- **Enumeration**
  - you can define a set
  - each member is represented as an integer

- **Preprocessor directives**
  - process your program before it is compiled
Structures

- Like enum, it may define a new type
- Aggregate variables of different types
- Each member of a structure can be
  - Array, structure, arrays of structures
- You can build an array of structures

```c
struct {
    int day, month, year;
    char day_name[4]; /* Mon, Tue, Wed, etc. */
    char month_name[4]; /* Jan, Feb, Mar, etc. */
} yesterday, today, tomorrow;

struct date {
    int day, month, year;
    char day_name[4]; /* Mon, Tue, Wed, etc. */
    char month_name[4]; /* Jan, Feb, Mar, etc. */
};
struct date yesterday, today, tomorrow;
```
Initialization

- Similar to the array initialization, but uses values of different types.

```c
struct date{
    int    day, month, year;
    char   day_name[4]; /* Mon, Tue, Wed, etc. */
    char   month_name[4]; /* Jan, Feb, Mar, etc. */
};

struct date final = {15, 12, 2014, "Mon", "Dec"};

struct date remaining_classes[3] =
    {
        {8, 12, 2014, "Mon", "Dec"},
        {9, 12, 2014, "Tue", "Dec"},
        {15, 12, 2014, "Mon", "Dec"}
    }
```
Accessing a Member (1)

- **Dot (.) operator**
  
  - `structure_name.member_name`

```c
struct {
    int    day, month, year;
    char   day_name[4]; /* Mon, Tue, Wed, etc. */
    char   month_name[4]; /* Jan, Feb, Mar, etc. */
} final;

...

final.day = 15;
final.month = 12;
final.year = 2014;
strcpy(final.day_name, "Mon");
strcpy(final.month_name, "Dec");
```
Accessing a Member (1)

- `->` operator
  - `pointer_to_structure->member_name`
  - `(*pointer_to_structure).member_name`

```c
struct {
    int   day, month, year;
    char  day_name[4]; /* Mon, Tue, Wed, etc. */
    char  month_name[4]; /* Jan, Feb, Mar, etc. */
} final, *final_p = &final;

...

final_p->day = 15;
final_p->month = 12;
(*final_p).year = 2014;
strcpy(final_p->day_name, "Mon");
strcpy(final_p->month_name, "Dec");
```
### Declaration and Assignment

```c
struct student tmp, *p = &tmp;
tmp.grade = 'A';
tmp.last_name = "Casanova";
tmp.student_id = 910017;
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Equivalent expression</th>
<th>Conceptual value</th>
</tr>
</thead>
<tbody>
<tr>
<td>tmp.grade</td>
<td>p-&gt;grade</td>
<td>A</td>
</tr>
<tr>
<td>tmp.last_name</td>
<td>p-&gt;last_name</td>
<td>Casanova</td>
</tr>
<tr>
<td>(*p).student_id</td>
<td>p-&gt;student_id</td>
<td>910017</td>
</tr>
<tr>
<td>*p-&gt;last_name + 1</td>
<td>(*p-&gt;last_name) + 1</td>
<td>'D'</td>
</tr>
<tr>
<td>*(p-&gt;last_name+2)</td>
<td>*(p-&gt;last_name)[2]</td>
<td>'s'</td>
</tr>
</tbody>
</table>
Scope Rule: Structure

- Similar to the scope rules of variables

```c
int foo(struct date) {
    ...
    invisible
}

int main(void) {
    struct date {
        int d, m, y;
    } final;
    foo(final) ...
}

struct date {
    int d, m, y;
};

int foo(struct date) {
    ...
}

int main(void) {
    struct final;
    foo(final) ...
}
```
Using structures

- Assignment works as long as two variables are of the same structure type
  
  ```
  struct date a, b;
  a = b;
  ```

- Structure is more like a primitive type when used as a function parameter

  - Call by value – the whole structure is copied
    - If it contains an array, the whole array is copied
    - Inefficient
    - This is one of reasons why there exists the -> operator
  
  - Call by reference – pointer is copied
Structure: Call-by-Value

- Copies whole data in a structure

```c
struct date update_day(struct date date) {
    scanf("%d %d %d %s %s", &date.day, &date.mon, 
            &date.year, date.day_name, date.mon_name);
    return date;
}

int main(void) {
    struct date a = {1, 1, 2014, "Mon", "Jan"};
    a = update_day(a);
}
```

Program

```
update_day()
  date:

main()
  a:
```

- copy for function call
- copy for return
Structure: Call-by-Reference

- Copies reference (address) of a structure

```c
void update_day(struct date *date) {
    scanf("%d %d %d %s %s", &date->day, &date->mon, 
            &date->year, date->day_name, date->mon_name);
    return;
}
int main(void) {
    struct date a = {1, 1, 2014, "Mon", "Jan"};
    update_day(&a);
}
```
UNION
Unions

- Similar to structure, but
- Defines a set of alternative values that may be stored in a shared location
- Programmer is responsible for interpreting the value correctly

```c
union int_or_double {
    int i;
    double d;
} a;
```
Unions

- To access a union member
  - . operator
  - -> operator

- The members of a structure and or a union can be array, structure, union

- Assignment, call-by-value, call-by-reference of a union can be done like a structure
#include <stdio.h>

typedef union int_or_float {
    int     i;
    float   f;
} number;

int main(void)
{
    number   n;
    
    n.i = 4444;
    printf("i: %10d     f: %16.10e\n", n.i, n.f);
    n.f = 4444.0;
    printf("i: %10d     f: %16.10e\n", n.i, n.f);
    return 0;
}
A bit field is an **int** or **unsigned** member of a **structure** or a **union**

- Use . operator or -> operator to access members

**Bit fields may be unnamed**

- Unnamed bit field of width 0 is for alignment of the next word

**Restrictions**

- Array of bit fields
- Address operator &
```c
#include <limits.h>
#include <stdio.h>

typedef struct {
    unsigned   b0 : 8, b1 : 8, b2 : 8, b3 : 8;
} word_bytes;

typedef struct {
    unsigned  
        b0 : 1, b1 : 1, b2 : 1, b3 : 1, b4 : 1, b5 : 1, b6 : 1,
        b7 : 1, b8 : 1, b9 : 1, b10 : 1, b11 : 1, b12 : 1, b13 : 1,
        b14 : 1, b15 : 1, b16 : 1, b17 : 1, b18 : 1, b19 : 1, b20 : 1,
        b21 : 1, b22 : 1, b23 : 1, b24 : 1, b25 : 1, b26 : 1, b27 : 1,
        b28 : 1, b29 : 1, b30 : 1, b31;
} word_bits;

typedef union {
    int         i;
    word_bits   bit;
    word_bytes  byte;
} word;

word w = {0};
    w.i = 'A';
    putchar(w.i);
    w.bit.b6 = 1; w.bit.b0 = 1;
    putchar(w.i);
```