

Structures and Unions

Young W. Lim

2020-10-08 Thr

- 1 Structures and unions
 - Based on
 - Structure Background
 - Union Background

- 1 "Self-service Linux: Mastering the Art of Problem Determination",

Mark Wilding

- 1 "Computer Architecture: A Programmer's Perspective", Bryant & O'Hallaron

I, the copyright holder of this work, hereby publish it under the following licenses: GNU head Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled GNU Free Documentation License.

CC BY SA This file is licensed under the Creative Commons Attribution ShareAlike 3.0 Unported License. In short: you are free to share and make derivative works of the file under the conditions that you appropriately attribute it, and that you distribute it only under a license compatible with this one.

Compiling 32-bit program on 64-bit gcc

- `gcc -v`
- `gcc -m32 t.c`
- `sudo apt-get install gcc-multilib`
- `sudo apt-get install g++-multilib`
- `gcc-multilib`
- `g++-multilib`
- `gcc -m32`
- `objdump -m i386`

Structures (1)

- **structures**
 - combining objects of different types
- **unions**
 - aggregate multiple objects into a single unit
 - allows an objects to be referenced using several different types

Structures (2)

- group objects possible different types into a single object
- like arrays
 - stored in a contiguous region
 - a pointer to a structure : the address of its 1st byte
- compiler maintains information about each structure type indicating the byte offset of each field
- compiler generates references to structure elements using these offset as displacements in memory referencing instructions

Rectangle Structure Example (1)

- to represent a rectangle as a structure

```
struct rect {  
    int llx;        // x coordinate of lower-left corner  
    int lly;        // y coordinate of lower-left corner  
    int color;      // coding of color  
    int width;      // width (in pixels)  
    int height;     // height (in pixels)  
};
```

- to declare a structure variable `r`

```
struct rect r;
```

- to access fields of a structure variable `r`

```
r.llx = r.lly = 0;  
r.color = 0xFF00FF;  
r.width = 10;  
r.height = 20;
```

Rectangle Structure Example (2)

- to represent a rectangle as a structure

```
struct rect {  
    int llx;        // x coordinate of lower-left corner  
    int lly;        // y coordinate of lower-left corner  
    int color;      // coding of color  
    int width;      // width (in pixels)  
    int height;     // height (in pixels)  
};
```

- to compute the area of a rectangle

```
int area (struct rect *rp)  
{  
    return (*rp).width * (*rp).height;  
}
```


Rectangle Structure Example (3)

- to represent a rectangle as a structure

```
struct rect {
    int llx;        // x coordinate of lower-left corner
    int lly;        // y coordinate of lower-left corner
    int color;      // coding of color
    int width;      // width (in pixels)
    int height;     // height (in pixels)
};
```

- to rotate a rectangle

```
void rotate_left (struct rect *rp)
{ // swap width and height
    int t          = rp->height;
    rp->height = rp->width;
    rp->width  = t;
    return (*rp).width * (*rp).height;
}
```

Structure fields accessing Example (1)

```
struct rec {
    int i;        // 4 bytes
    int j;        // 4 bytes
    int a[3];    // 12 bytes
    int *p;       // 4 bytes
}

0x00 : i
0x04 : j
0x08 : a[0]
0x0C : a[1]
0x10 : a[2]
0x14 : p
0x1C :
```

offset	0	4	8	12	16
contents	i	j	a[0]	a[1]	a[2]
size	4 bytes	4 bytes	4 bytes	4 bytes	4 bytes

Strudcture Exmapple (4)

```
movl    (%edx), %eax    ; Get r->i
movl    %eax, 4(%edx)   ; Store in r->j

; r in %eax, i in %edx
leal   8(%eax, %edx, 4) ; %ecx = &r->a[i]
```

Strudcture Exmapple (5)

```
r->p = &r->[r->i + r->j];
```

```
movl 4(%edx), %eax      ; Get r-j
addl (%edx), %eax      ; Add r-i
leal 8(%edx, %eax, 4), %eax ; Compute &r->[r->i + r->j]
movl %eax, 20(%edx)     ; Store in r->p
```

Strudcture Exmample (6)

```
struct prob {  
    int *p;  
    struct {  
        int x;  
        int y;  
    } s;  
    struct prob *next;  
};
```

```
movl 8(%ebp), %eax  
movl 8(%eax), %edx  
movl %edx, 4(%eax)  
leal 4(%eax), %eax  
movl %edx, (%eax)  
movl %eax, 12(%eax)
```

Structure Declaration (2)

- `struct rec *r;`
- copy the element of `r->i` to element `r->j`

`r->j = r->i`

```
movl (%edx), %eax      ; Get r->i
movl %eax, 4(%edx)     ; Store in r->j
```

Structure Declaration (3)

- `struct rec *r;`
- to generate a pointer to an object within a structure simply add the field's offset to the structure address
 - generate the pointer `&(r->a[i])` by adding offset $8 + 4 \cdot 1 = 12$
 - for pointer `r` in register `%eax`
integer variable `i` in register `%edx`

`r` in `%eax`, `i` in `%edx`

```
leal 8(%eax, %edx, 4), %ecx ; %ecx = &r->a[i]
```

Structure Declaration (4)

- `struct rec *r;`
- `r->p = &r->a[r->i + r->j];`
- ```
movl 4(%edx), %eax ; get r->j
addl (%edx), %eax ; add r->i
leal 8(%edx, %eax, 4), %eax ; compute &r->[r->i + r->j]
movl %eax, 20(%edx) ; store in r->p
```



# Unions (1)

- structures
  - combining objects of different types
- **unions**
  - aggregate multiple objects into a single unit
  - allows an objects to be referenced using several different types

## Unions (2)

- allow a single object to be referenced according to multiple types
- the syntax of a union declaration is identical to that for structures
- the different semantics
- rather than having the different fields reference different blocks
- but they all reference the same block
- the use of two different fields is mutually exclusive
- can reduce memory usage<sup>3</sup>
- can be used to access the bit patterns of different data types

# Union Declaration (1)

```
struct S3 {
 char c;
 int i[2];
 double v;
};
```

```
0x00 : c
0x04 : i[0]
0x08 : i[1]
0x0c : v
0x20 :
```

```
union U3 {
 char c;
 int i[2];
 double v;
}
```

```
0x00 : c
0x00 : i[0]
0x00 : i[1]
0x00 : v
0x08 :
```

## Union Declaration (2)

```
struct NODE {
 struct NODE *left;
 struct NODE *right;
 double data;
};
```

```
union NODE{
 struct NODE {
 struct NODE *left;
 struct NODE *right;
 } internal;
 double data;
};
```

```
struct NODE {
 int is_leaf;
 union NODE{
 struct NODE {
 struct NODE *left;
 struct NODE *right;
 } internal;
 double data;
 } info;
};
```

# Union Declaration (3)

```
unsigned float2bit(float f)
{
 union {
 float f;
 unsigned u;
 } temp;
 temp.f = f;
 return temp.u;
};
```

```
unsigned copy(unsigned u)
{
 return u;
}

movl 8(%ebp), %eax
```

## Union Declaration (4)

```
double bit2double(unsigned word0, unsigned word1)
{
 union {
 double d;
 unsigned u[2];
 } temp;

 temp.u[0] = word0;
 temp.u[1] = word1;
 return temp.d;
}
```