Structures and Unions

Young W. Lim

2021-12-01 Wed

Outline

Based on

1. "Self-service Linux: Mastering the Art of Problem Determination", Mark Wilding
1. "Computer Architecture: A Programmer's Perspective", Bryant & O'Hallaron

Compling 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

Definitions of structures and unions

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

Characteristics of structures

- *combining* objects of *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 elements
  by indicating the byte offset of each field
- compiler generates references
  to structure elements
  using these byte offset as displacements
  in memory referencing instructions

Structure rect (1) declaring and setting

Representing 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)
};
```

Declaring r

```
struct rect r;
```

Setting fields r

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

Structure rect (2) passing a structure pointer

rect structure

```
struct rect {
  int llx;
  int lly;
  int color;
  int width;
  int height;
};
```

- a pointer rp to the rect structure is passed to the function area
- *rp.width = *(rp.width) : wrong
- (*rp).width = rp->width

Computing the area of a rectangle

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

Structure rect (3)

rect structure

```
struct rect {
  int llx;
  int lly;
  int color;
  int width;
  int height;
};
```

- swap width and height fields
- rp : pointer to rect structure
- (*rp).width = rp->width
- (*rp).height = rp->height

Rotating 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 rec (1) memory layout

struct rec

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

}
```

memory layout

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

offset	0	4	8	12	16	20
contents	i	j	a[0]	a[1]	a[2]	p
size	4 bytes	4 bytes	4 bytes	4 bytes	4 bytes	4 bytes

Structure rec (2) get, store, address

struct pointer r

```
struct rec *r;

; load field r->i
movl   (%edx), %eax

; store at field r->j
movl   %eax, 4(%edx)
```

- r is in %edx
- copy r->i into r->j

offset of the array a

```
; &r->a[i] = &(r->a[i])
; %eax + 4*%edx + 8
; r    + 4*i    + a

leal   8(%eax, %edx, 4), %ecx
```

- r is in %eax
- i is in %edx
- array a has an offset 8

Structure rec (3) copy i field to j field

offset	0	4	8	20
contents	i	j	a	p
register	(%edx)	4(%edx)	8(%edx)	20(%edx)

- 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 rec (4) address of a field

contents	a[0]	a[1]	a[2]
offset	$8 = 8 + 4 * 0$	$12 = 8 + 4 * 1$	$16 = 8 + 4 * 2$
%eax	r	r	r
%edx	0	1	2

- pointer r in register %eax
- integer variable i in register %edx

%ecx = &r->a[i]

```
leal 8(%eax, %edx, 4), %ecx
```

Structure rec (5) address of a field

contents	a[0]	a[1]	a[2]
offset	$8 = 8 + 4 * 0$	$12 = 8 + 4 * 1$	$16 = 8 + 4 * 2$
%eax	r	r	r
%edx	0	1	2

- to generate a pointer to an object within a structure
  simply add the field's offset ($8 + 4 \cdot$ %edx)
  to the structure address r (%eax)
  - generate the pointer &(r->a[i])
    by adding offset $8 + 4 \cdot 1 = 12$

%ecx = &r->a[i]

```
leal 8(%eax, %edx, 4), %ecx
```

Structure rec (6)

struct rec

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

}
```

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->a[r->i + r->j]
movl  %eax, 20(%edx)          ; Store in r->p
```

Structure rec (7)

offset	0	4	8	20
contents	i	j	a	p
register	(%edx)	4(%edx)	8(%edx)	20(%edx)

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

Structure prob (1)

offset	0	4	8	12
contents	p	s.x	s.y	next
type	int	int	int	pointer
size	4 bytes	4 bytes	4 bytes	4 bytes
register	(%eax)	4(%eax)	8(%eax)	12(%eax)

source code

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

Structure prob (2)

source code

```
void sp_init(struct prob *sp) {
  sp->s.x =
  sp->p =
  sp->next =
}
```

assembly code

```
movl  8(%ebp), %eax   ; p => %eax
movl  8(%eax), %edx   ; s.y => %edx
movl  %edx, 4(eax)    ; s.y => s.x
leal  4(%eax), %edx   ; &s.x => %edx
movl  %edx, (%eax)    ; &s.x => p
movl  %eax, 12(%eax) ; p => next
```

Definitions of structures and unions

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

Unions (1)

- access a *single object* according to *multiple types*
- the same syntax of a union declaration as that for structures
- the different semantics

- rather than having the different fields reference different blocks
- but they all reference the same block

Unions (2)

- the mutually exclusive use of two different fields
  - can reduce memory usage
  - can be used to access the bit patterns of different data types

Union Declaration (1)

Structure

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

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

size = 20 byte
```

Union

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

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

size = 8 bytes
```

Union Declaration (2)

Structure

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

Union

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

field	c	i[0]	i[1]	v	size
type	char	int	int	double	
	1	4	4	8	
S3 offset	0	4	8	12	20
U3 offset	0	0	0	0	8

Union Declaration (2')

Structure

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

Union

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

	0x7	0x6	0x5	0x4	0x3	0x2	0x1	0x0
c								<xxx>
i[0]					<xxxx	xxxxx	xxxxx	xxxx>
i[1]	<xxxx	xxxxx	xxxxx	xxxx>				
v	<xxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxx>

Union Declaration (2")

- `i` has offset 4 in `S3` rather than 1 (alignment)
- for pointer `p` of type `union U3*`
  references `p->c`, `p->i[0]`, `p->v`
  would all reference the beginning of the data structure
- the overall size of a union equals
  the maximum size of any of its fields

Structure

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

Union

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

Union Declaration (3)

- to implement a binary tree data structure
  where each leaf node has a double data value,
  while each internal node has pointers ot two children

Structure

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

4 + 4 + 8 = 16 bytes

Union

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

4 + 4 = 8 bytes

Union Declaration (4)

- if `n` is a pointer to a node of type union `NODE *` we would reference the data of a leaf node as `n->data`, and the children of an internal node as `n->internal.left` and `n->internal.right`

Union Dclaration (5)

- there is no way to determine
  whether a given node is leaf or an internal node
- a common way is to introduce an additional tag field `is_leaf`
  - `is_leaf` is 1 for a leaf node
  - 0 for an internal node

Structure

```
struct NODE {
  int is_leaf;                // 4 bytes
  union NODE{
    struct NODE {
      struct NODE *left;      // 4 bytes
      struct NODE *right;     // 4 bytes
    } internal;               // 8 bytes
    double data;              // 8 bytes
  } info;                     // 8 bytes
};                            // 12 bytes
```

Union Dclaration (6)

- this structure requires 12 bytes
  - 4 bytes for `is_leaf`
  - 4 bytes for `info.internal.left` or `info.internal.right`
  - 8 bytes for `info.data`

Structure

```
struct NODE {
  int is_leaf;                  // 4 bytes
  union NODE{
    struct NODE {
      struct NODE *left;        // 4 bytes
      struct NODE *right;       // 4 bytes
    } internal;                 // 8 bytes
    double data;                // 8 bytes
  } info;                       // 8 bytes
};                              // 12 bytes
```

Union Dclaration (7)

- in this case, the savings gain of using a union is small relative to the awkwardness of the resulting code
- for data structures with more fields, the savings can be more compelling

Union Declaration (8)

- unions can also be used to access the bit patterns of different data types
- the following code returns the bit representation of a `float` as an `unsigned`

`float2bit(float f)`

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

Union Declaration (9)

- in this code, we store the argument in the union using one data type, and access it using another
- Interestingly, the code generated for this procedure is identical to that for the following procedure;

copy

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

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

Union Declaration (10)

- the body of both procedure is just a single instruction

  `movl 8(%ebp), %eax`

- this demonstrates the lack of type information
  in assembly code
- the argument will be at offset 8 relative to %ebp
  regardless of whether it is a `float` or an `unsigned`
- the procedure simply copies its argument as the return value
  without modifying any bits

Union Declaration (11)

- when using unions to combine data types of different sizes, byte ordering issues can become important
- for example, suppose we write a procedure that will create an 8-byte `double` using the bit patterns given by two 4-byte `unsigned`'s

bit2double

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

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

Union Declaration (12)

- on a little endian machine such as IA32, argument `word0` will become the low order four bytes of `d`
  while `word1` will become the high order four bytes
- on a big endian machine, the role of the two arguments will be reversed

bit2double

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

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

Union Declaration (13)

- unions can be useful in several contexts
  however, the can also load to nasty bugs,
  since they bypass the safety provided by the c type system

- one application is when we know in advance
  that the use of two different fields in a data structure
  will be mutually exclusive

- then declaring these tow fields as part of a union
  rather than a structure will reduce the total space allocated