Structure.20220101

Young W. Lim

2022-01-01 Sat

Outline

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

'#+TITLE: Structures and Unions

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

Structure S3 and union U3 (1)

Structure S3

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

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

size = 20 byte
```

Union U3

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

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

size = 8 bytes
```

Structure S3 and union U3 (2)

Structure S3

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

Union U3

```
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

Structure S3 and union U3 (3)

Offset and size

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

size = 8 bytes
```

Union U3

```
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>

Structure S3 and union U3 (4)

- i has offset 4 in S3 rather than 1 (alignment)
  - c has the size of one byte
  - 3 bytes are padded

Structure S3

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

Union U3

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

Structure S3 and union U3 (5)

- `union U3 *p`
  - `p->c`, `p->i[0]`, `p->v` would all reference the beginning of the data structure (offset 0)
- the overall size of a union equals to the maximum size of any of its fields
  - `sizeof(U3) = sizeof(i)` = 8 bytes

Structure S3

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

Union U3

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

Binary Tree Data Structure

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

Structure NODE

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

```
4 + 4 + 8 = 16 bytes
```

Union NODE

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

```
4 + 4 = 8 bytes
```

Accessing leaf and internal nodes

- if `n` is a pointer to a node of type union `NODE *`

Union NODE

```
union NODE{

  struct NODE {
    struct NODE *left;
    struct NODE *right;
  } internal;

  double data;
};
```

accessing fields

- `union NODE *n`
- the data of a leaf node
  `n->data`
- the children of an internal node
  `n->internal.left`
  `n->internal.right`

Using a tag field

- use a tag field is_leaf
  to determine whether a given node is leaf or an internal node
  - is_leaf = 1 : for a leaf node
  - is_leaf = 0 : for an internal node

Structure NODE with a tag

```
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
```

Sizes of structure NODE with a tag field

- this structure requires 12 bytes

is_leaf	int	4 bytes
info.internal.left	pointer	4 bytes
info.internal.right	pointer	4 bytes
info.data	double	8 bytes

Structure NODE with a tag

```
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
```

Savings gain of union

- without a union
  - each leaf node wastes 8 byte `data` field
  - each internal node wastes two 4 byte pointer fields
- for data structures with more fields,
  the savings can be more compelling

Structure NODE

```
struct NODE {
  struct NODE *left;    // 4 bytes
  struct NODE *right;   // 4 bytes
  double data;          // 8 bytes
};
```

4 + 4 + 8 = 16 bytes

Structure NODE with a tag

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

Union usages

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

- when we know in advance that the use of two different fields
  will be mutually exclusive
  then declaring these two fields as part of a union
  rather than a structure will reduce the total space allocated

Accessing different data types (1)

- unions can also be used to access the bit patterns of different data types
- code for returning the bit representation of a `float` (4 bytes) as an `unsigned` (1 byte)

float2byte(float f)

```
unsigned float2byte(float f)
{
  union {
    float f;       // 4 bytes
    unsigned u;    // 1 byte + 3 padding bytes
  } temp;

  temp.f = f;      // 4 bytes
  return temp.u;   // LSByte
};
```

Accessing different data types (2)

- in `float2byte`, store the argument in the union using one data type, and access it using another
- the code generated for this procedure is identical to that for copy;

float2byte(float f)

```
 unsigned float2bit(float f)
{
   union {
     float f;        // 4 bytes
     unsigned u;     // 1 byte
                     // 3 padding
   } temp;

   temp.f = f;       // 4 bytes
   return temp.u;    // LSByte
 };
```

copy

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




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

Accessing different data types (3)

- the body of both `float2byte` and `copy` 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 `8(%ebp)` as the return value `%eax` without modifying any bits

Byte ordering (1)

- when using unions to combine data types of different sizes, byte ordering issues can become important
- create an 8-byte `double` using the bit patterns given by two 4-byte `unsigned`'s

byte2double

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

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

Byte ordering (2)

- 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

	0x7	0x6	0x5	0x4	0x3	0x2	0x1	0x0
`u[0]`					<xxxx	xxxxx	xxxxx	xxxx>
`u[1]`	<xxxx	xxxxx	xxxxx	xxxx>				
`d`	<xxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxx>

Union ele (1)

offset	0	4	0	4
contents	`e1.p`	`e1.y`	`e2.x`	`e2.next`
type	pointer	int	int	pointer
size	4 bytes	4 bytes	4 bytes	4 bytes
register				

source code

```
union ele {
  struct {
    int *p;
    int  y;
  } e1;
  struct {
    int  x;
    union ele *next;
  } e2;
};
```

Union ele (2)

- procedure operates on a linked list having these unions as elements
- some union references can have ambiguous interpretations
- these ambiguities get resolved as you see where the references lead
- there is only one answer that does not perform any casting
  and does not violate any type constraints

source code

```
void proc (union ele *up)
{
  up->   = up -> - up ->
}
```

Union ele (3)

- procedure operates on a linked list having these unions as elements

source code

```
void proc (union ele *up)
{
  up->   = up -> - up ->
}
```

assembly code

```
movl 8(%ebp), %eax   ; up => %eax
movl 4(%eax), %edx   ; up->y => %edx
movl (%edx), %ecx
movl %ebp, %esp
movl (%eax), %eax
movl (%ecx), %ecx
subl %eax, %ecx
movl %ecx, 4(%edx)
```