Structures and Unions

Young W. Lim

2020-10-21 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

Strudctures (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

Strudctures (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 Exmaple (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 Exmaple (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 Exmaple (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 rotage 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;
}
```

Strudcture fields accessing Exmaple (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 Exmaple (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 Exmaple (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 Exmaple (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(Teax)
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 addthe 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 <u>different</u> 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 mulitple 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 usage3
- 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 :

size = 20 bytes
```

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

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

size = 8 bytes
```

Union Declaration (2)

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

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

type	c	i	v	size
S3	0	4	12	20
U3	0	0	0	8

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

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

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

4 + 4 + 8 = 16 bytes

4 + 4 = 8 bytes

Union Dclaration (4)

- 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

```
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 (5)

- 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`

```
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)

- 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 (7)

- 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`

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

Union Declaration (8)

- 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;

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

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

Union Declaration (9)

- 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 (10)

- 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

```
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 (11)

- 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

```
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)

- 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